Document Number:	Dxxxx
Date:	2017-03-17
Revises:	N4656
Reply to:	Jonathan Wakely
	cxx@kayari.org

Working Draft, C++ Extensions for Networking

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

List of Tables

1 Scope [scope]

1 This Technical Specification describes extensions to the C++ Standard Library. This Technical Specification specifies requirements for implementations of an interface that computer programs written in the C++ programming language may use to perform operations related to networking, such as operations involving sockets, timers, buffer management, host name resolution and internet protocols. This Technical Specification is applicable to information technology systems that can perform network operations, such as those with operating systems that conform to the POSIX interface. This Technical Specification is applicable only to vendors who wish to provide the interface it describes.

2 Normative references [references]

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2 [*Note:* The programming language and library described in ISO/IEC 14882 is herein called the C++ Standard. References to clauses within the C++ Standard are written as "C++Std [library]". The operating system interface described in ISO/IEC 9945 is herein called POSIX. — *end note*]

3 This Technical Specification mentions commercially available operating systems for purposes of exposition. POSIX® is a registered trademark of The IEEE. Windows® is a registered trademark of Microsoft Corporation. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of these products.

4 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++Std [library]) is included into this Technical Specification by reference.

3 Terms and definitions [defs]

3.1 **[defs.host.byte.order]**
host byte order
see section 3.194 of POSIX Base Definitions, Host Byte Order

3.2 **[defs.net.byte.order]**
network byte order
see section 3.238 of POSIX Base Definitions, Network Byte Order

3.3 **[defs.sync.op]**
synchronous operation
an operation where control is not returned until the operation completes

3.4 **[defs.async.op]**
asynchronous operation
an operation where control is returned immediately without waiting for the operation to complete [*Note:* Multiple asynchronous operations may be executed concurrently. — *end note*]

3.5 **[defs.orderly.shutdown]**
orderly shutdown
the procedure for shutting down a stream after all work in progress has been completed, without loss of data

4 General Principles [general]

4.1 Conformance [conformance]

1 Conformance is specified in terms of behavior. Ideal behavior is not always implementable, so the conformance sub-clauses take that into account.

4.1.1 POSIX conformance [conformance.9945]

1 Some behavior is specified by reference to POSIX. How such behavior is actually implemented is unspecified.

2 [*Note:* This constitutes an "as if" rule allowing implementations to call native operating system or other APIs. — *end note*]

3 Implementations are encouraged to provide such behavior as it is defined by POSIX. Implementations shall document any behavior that differs from the behavior defined by POSIX. Implementations that do not support exact POSIX behavior are encouraged to provide behavior as close to POSIX behavior as is reasonable given the limitations of actual operating systems and file systems. If an implementation cannot provide any reasonable behavior, the implementation shall report an error as specified in Error Reporting (9).

4 [*Note:* This allows users to rely on an exception being thrown or an error code being set when an implementation cannot provide any reasonable behavior. — *end note*]

5 Implementations are not required to provide behavior that is not supported by a particular operating system.

4.1.2 Conditionally-supported features [conformance.conditional]

1 This Technical Specification defines conditionally-supported features, in the form of additional member functions on types that satisfy `Protocol` (18.2.6), `Endpoint` (18.2.4), `SettableSocketOption` (18.2.9), `GettableSocketOption` (18.2.8) or `IoControlCommand` (18.2.12) requirements.

2 [*Note:* This is so that, when the additional member functions are available, C++ programs may extend the library to add support for other protocols and socket options. — *end note*]

3 For the purposes of this Technical Specification, implementations that provide all of the additional member functions are known as extensible implementations.

4 [*Note:* Implementations are encouraged to provide the additional member functions, where possible. It is intended that POSIX and Windows implementations will provide them. — *end note*]

4.2 Acknowledgments [intro.ack]

1 The design of this specification is based, in part, on the Asio library written by Christopher Kohlhoff.

5 Namespaces and headers [namespaces]

1 The components described in this Technical Specification are experimental and not part of the C++ standard library. All components described in this Technical Specification are declared in namespace `std::experimental::net::v1` or a sub-namespace thereof unless otherwise specified. The headers described in this technical specification shall import the contents of `std::experimental::net::v1` into `std::experimental::net` as if by:

```
namespace std {
  namespace experimental {
    namespace net {
      inline namespace v1 {}
    }
  }
}
```

2 Unless otherwise specified, references to other entities described in this Technical Specification are assumed to be qualified with `std::experimental::net::v1::`, references to entities described in the C++ standard are assumed to be qualified with `std::`, and references to entities described in C++ Extensions for Library Fundamentals are assumed to be qualified with `std::experimental::fundamentals_v1::`.

6 Future plans (Informative) [plans]

1 This section describes tentative plans for future versions of this technical specification and plans for moving content into future versions of the C++ Standard.

2 The C++ committee may release new versions of this technical specification, containing networking library extensions we hope to add to a near-future version of the C++ Standard. Future versions will define their contents in `std::experimental::net::v2`, `std::experimental::net::v3`, etc., with the most recent implemented version inlined into `std::experimental::net`.

3 When an extension defined in this or a future version of this technical specification represents enough existing practice, it will be moved into the next version of the C++ Standard by replacing the `experimental::net::vN` segment of its namespace with `net`, and by removing the `experimental/` prefix from its header's path.

7 Feature test macros (Informative) [feature.test]

1 These macros allow users to determine which version of this Technical Specification is supported by the headers defined by the specification. All headers in this Technical Specification shall define the `__cpp_lib_experimental_net` feature test macro in Table 1.

2 If an implementation supplies all of the conditionally-supported features specified in 4.1.2, all headers in this Technical Specification shall additionally define the `__cpp_lib_experimental_net_extensible` feature test macro.

Table 1 — Feature-test macro(s)

Macro name	Value
`__cpp_lib_experimental_net`	201602
`__cpp_lib_experimental_net_extensible`	201602

8 Method of description (Informative) [description]

1 This sub-clause describes the conventions used to specify this Technical Specification, in addition to those conventions specified in C++Std [description].

8.1 Structure of each clause [structure]

8.1.1 Detailed specifications [structure.specifications]

1 In addition to the elements defined in C++Std [structure.specifications], descriptions of function semantics contain the following elements (as appropriate):

(1.1) — *Completion signature:* if the function initiates an asynchronous operation, specifies the signature of a completion handler used to receive the result of the operation.

8.2 Other conventions [conventions]

8.2.1 Nested classes [nested.class]

1 Several classes defined in this Technical Specification are nested classes. For a specified nested class `A::B`, an implementation is permitted to define `A::B` as a synonym for a class with equivalent functionality to class `A::B`. [*Note:* When `A::B` is a synonym for another type `A` shall provide a nested type `B`, to emulate the injected class name. — *end note*]

9 Error reporting [err.report]

9.1 Synchronous operations [err.report.sync]

1 Most synchronous network library functions provide two overloads, one that throws an exception to report system errors, and another that sets an `error_code` (C++Std [syserr]).

[*Note:* This supports two common use cases:

(1.1) — Uses where system errors are truly exceptional and indicate a serious failure. Throwing an exception is the most appropriate response.

(1.2) — Uses where system errors are routine and do not necessarily represent failure. Returning an error code is the most appropriate response. This allows application specific error handling, including simply ignoring the error.

— *end note*]

2 Functions not having an argument of type `error_code&` report errors as follows, unless otherwise specified:

(2.1) — When a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, the function exits via an exception of a type that would match a handler of type `system_error`.

(2.2) — Destructors throw nothing.

3 Functions having an argument of type `error_code&` report errors as follows, unless otherwise specified:

(3.1) — If a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, the `error_code&` argument `ec` is set as appropriate for the specific error. Otherwise, the `ec` argument is set such that `!ec` is `true`.

4 Where a function is specified as two overloads, with and without an argument of type `error_code&`:

```
R f(A1 a1, A2 a2, ..., AN aN);
R f(A1 a1, A2 a2, ..., AN aN, error_code& ec);
```

5 then, when R is non-`void`, the effects of the first overload are as if:

```
error_code ec;
R r(f(a1, a2, ..., aN, ec));
if (ec) throw system_error(ec, S);
return r;
```

6 otherwise, when R is `void`, the effects of the first overload are as if:

```
error_code ec;
f(a1, a2, ..., aN, ec);
if (ec) throw system_error(ec, S);
```

7 except that the type thrown may differ as specified above. `S` is an NTBS indicating where the exception was thrown. [*Note:* A possible value for `S` is `__func__`. — *end note*]

8 For both overloads, failure to allocate storage is reported by throwing an exception as described in the C++ standard (C++Std [res.on.exception.handling]).

9 In this Technical Specification, when a type requirement is specified using two function call expressions *f*, with and without an argument `ec` of type `error_code`:

```
f(a1, a2, ..., aN)
f(a1, a2, ..., aN, ec)
```

then the effects of the first call expression of *f* shall be as described for the first overload above.

9.2 Asynchronous operations [err.report.async]

1 Asynchronous network library functions in this Technical Specification are identified by having the prefix `async_` and take a completion handler 13.2.7.2. These asynchronous operations report errors as follows:

(1.1) — If a call by the implementation to an operating system or other underlying API results in an error that prevents the asynchronous operation from meeting its specifications, the completion handler is invoked with an `error_code` value `ec` that is set as appropriate for the specific error. Otherwise, the `error_code` value `ec` is set such that `!ec` is `true`.

(1.2) — Asynchronous operations shall not fail with an error condition that indicates interruption of an operating system or underlying API by a signal. [*Note:* Such as POSIX error number `EINTR` — *end note*] Asynchronous operations shall not fail with any error condition associated with non-blocking operations. [*Note:* Such as POSIX error numbers `EWOULDBLOCK`, `EAGAIN`, or `EINPROGRESS`; Windows error numbers `WSAEWOULDBLOCK` or `WSAEINPROGRESS` — *end note*]

2 In this Technical Specification, when a type requirement is specified as a call to a function or member function having the prefix `async_`, then the function shall satisfy the error reporting requirements described above.

9.3 Error conditions [err.report.conditions]

1 Unless otherwise specified, when the behavior of a synchronous or asynchronous operation is defined "as if" implemented by a POSIX function, the `error_code` produced by the function shall meet the following requirements:

(1.1) — If the failure condition is one that is listed by POSIX for that function, the `error_code` shall compare equal to the error's corresponding `enum class errc` (C++Std [syserr]) or `enum class resolver_errc` (21.3) constant.

(1.2) — Otherwise, the `error_code` shall be set to an implementation-defined value that reflects the underlying operating system error.

2 [*Example:* The POSIX specification for `shutdown` lists `EBADF` as one of its possible errors. If a function that is specified "as if" implemented by `shutdown` fails with `EBADF` then the following condition holds for the `error_code` value `ec`: `ec == errc::bad_file_descriptor` — *end example*]

3 When the description of a function contains the element Error conditions, this lists conditions where the operation may fail. The conditions are listed, together with a suitable explanation, as `enum class` constants. Unless otherwise specified, this list is a subset of the failure conditions associated with the function.

9.4 Suppression of signals [err.report.signal]

1 Some POSIX functions referred to in this Technical Specification may report errors by raising a `SIGPIPE` signal. Where a synchronous or asynchronous operation is specified in terms of these POSIX functions, the generation of `SIGPIPE` is suppressed and an error condition corresponding to POSIX `EPIPE` is produced instead.

10 Library summary [summary]

Table 2 — Networking library summary

Clause	Header(s)
Convenience header (11)	`<experimental/net>`
Forward declarations (12)	`<experimental/netfwd>`
Asynchronous model (13)	`<experimental/executor>`
Basic I/O services (14)	`<experimental/io_context>`
Timers (15)	`<experimental/timer>`
Buffers (16) Buffer-oriented streams (17)	`<experimental/buffer>`
Sockets (18) Socket iostreams (19) Socket algorithms (20)	`<experimental/socket>`
Internet protocol (21)	`<experimental/internet>`

1 Throughout this Technical Specification, the names of the template parameters are used to express type requirements, as listed in Table 3.

Table 3 — Template parameters and type requirements

template parameter name	type requirements
`AcceptableProtocol`	acceptable protocol (18.2.7)
`Allocator`	C++Std [allocator.requirements]
`AsyncReadStream`	buffer-oriented asynchronous read stream (17.1.2)
`AsyncWriteStream`	buffer-oriented asynchronous write stream (17.1.4)
`Clock`	C++Std [time.clock.req]
`CompletionCondition`	completion condition (17.1.5)
`CompletionToken`	completion token (13.2.7.2)
`ConnectCondition`	connect condition (18.2.13)
`ConstBufferSequence`	constant buffer sequence (16.2.2)
`DynamicBuffer`	dynamic buffer (16.2.3)
`EndpointSequence`	endpoint sequence (18.2.5)
`ExecutionContext`	execution context (13.2.3)
`Executor`	executor (13.2.2)
`GettableSocketOption`	gettable socket option (18.2.8)
`InternetProtocol`	Internet protocol (21.2.1)
`IoControlCommand`	I/O control command (18.2.12)
`MutableBufferSequence`	mutable buffer sequence (16.2.1)
`ProtoAllocator`	proto-allocator (13.2.1)
`Protocol`	protocol (18.2.6)
`Service`	service (13.2.4)
`SettableSocketOption`	settable socket option (18.2.9)
`Signature`	signature (13.2.5)

Table 3 — Template parameters and type requirements (continued)

template parameter name	**type requirements**
`SyncReadStream`	buffer-oriented synchronous read stream (17.1.1)
`SyncWriteStream`	buffer-oriented synchronous write stream (17.1.3)
`WaitTraits`	wait traits (15.2.1)

11 Convenience header [convenience.hdr]

11.1 Header `<experimental/net>` synopsis [convenience.hdr.synop]

```
#include <experimental/executor>
#include <experimental/io_context>
#include <experimental/timer>
#include <experimental/buffer>
#include <experimental/socket>
#include <experimental/internet>
```

1 [*Note:* This header is provided as a convenience for programs so that they may access all networking facilities via a single, self-contained `#include`. — *end note*]

12 Forward declarations [fwd.decl]

12.1 Header <experimental/netfwd> synopsis [fwd.decl.synop]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class execution_context;
  template<class T, class Executor>
    class executor_binder;
  template<class Executor>
    class executor_work_guard;
  class system_executor;
  class executor;
  template<class Executor>
    class strand;

  class io_context;

  template<class Clock> struct wait_traits;
  template<class Clock, class WaitTraits = wait_traits<Clock>>
    class basic_waitable_timer;
  using system_timer = basic_waitable_timer<chrono::system_clock>;
  using steady_timer = basic_waitable_timer<chrono::steady_clock>;
  using high_resolution_timer = basic_waitable_timer<chrono::high_resolution_clock>;

  template<class Protocol>
    class basic_socket;
  template<class Protocol>
    class basic_datagram_socket;
  template<class Protocol>
    class basic_stream_socket;
  template<class Protocol>
    class basic_socket_acceptor;
  template<class Protocol, class Clock = chrono::steady_clock,
    class WaitTraits = wait_traits<Clock>>
      class basic_socket_streambuf;
  template<class Protocol, class Clock = chrono::steady_clock,
    class WaitTraits = wait_traits<Clock>>
      class basic_socket_iostream;

  namespace ip {

    class address;
    class address_v4;
    class address_v6;
    template<class Address>
      class basic_address_iterator;
    using address_v4_iterator = basic_address_iterator<address_v4>;
    using address_v6_iterator = basic_address_iterator<address_v6>;
```

```
      template<class Address>
        class basic_address_range;
      using address_v4_range = basic_address_range<address_v4>;
      using address_v6_range = basic_address_range<address_v6>;
      class network_v4;
      class network_v6;
      template<class InternetProtocol>
        class basic_endpoint;
      template<class InternetProtocol>
        class basic_resolver_entry;
      template<class InternetProtocol>
        class basic_resolver_results;
      template<class InternetProtocol>
        class basic_resolver;
      class tcp;
      class udp;

    } // namespace ip
  } // inline namespace v1
  } // namespace net
  } // namespace experimental
  } // namespace std
```

1 Default template arguments are described as appearing both in `<netfwd>` and in the synopsis of other headers but it is well-formed to include both `<netfwd>` and one or more of the other headers. [*Note:* It is the implementation's responsibility to implement headers so that including `<netfwd>` and other headers does not violate the rules about multiple occurrences of default arguments. — *end note*]

13 Asynchronous model [async]

13.1 Header <experimental/executor> synopsis [async.synop]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class CompletionToken, class Signature>
    class async_result;

  template<class CompletionToken, class Signature>
    struct async_completion;

  template<class T, class ProtoAllocator = allocator<void>>
    struct associated_allocator;

  template<class T, class ProtoAllocator = allocator<void>>
    using associated_allocator_t = typename associated_allocator<T, ProtoAllocator>::type;

  // get_associated_allocator:

  template<class T>
    associated_allocator_t<T> get_associated_allocator(const T& t) noexcept;
  template<class T, class ProtoAllocator>
    associated_allocator_t<T, ProtoAllocator>
      get_associated_allocator(const T& t, const ProtoAllocator& a) noexcept;

  enum class fork_event {
    prepare,
    parent,
    child
  };

  class execution_context;

  class service_already_exists;

  template<class Service> Service& use_service(execution_context& ctx);
  template<class Service, class... Args> Service&
    make_service(execution_context& ctx, Args&&... args);
  template<class Service> bool has_service(execution_context& ctx) noexcept;

  template<class T> struct is_executor;

  template<class T>
    constexpr bool is_executor_v = is_executor<T>::value;

  struct executor_arg_t { };
  constexpr executor_arg_t executor_arg = executor_arg_t();
```

```
template<class T, class Executor> struct uses_executor;

template<class T, class Executor>
  constexpr bool uses_executor_v = uses_executor<T, Executor>::value;

template<class T, class Executor = system_executor>
  struct associated_executor;

template<class T, class Executor = system_executor>
  using associated_executor_t = typename associated_executor<T, Executor>::type;

// get_associated_executor:

template<class T>
  associated_executor_t<T> get_associated_executor(const T& t) noexcept;
template<class T, class Executor>
  associated_executor_t<T, Executor>
    get_associated_executor(const T& t, const Executor& ex) noexcept;
template<class T, class ExecutionContext>
  associated_executor_t<T, typename ExecutionContext::executor_type>
    get_associated_executor(const T& t, ExecutionContext& ctx) noexcept;

template<class T, class Executor>
  class executor_binder;

template<class T, class Executor, class Signature>
  class async_result<executor_binder<T, Executor>, Signature>;

template<class T, class Executor, class ProtoAllocator>
  struct associated_allocator<executor_binder<T, Executor>, ProtoAllocator>;

template<class T, class Executor, class Executor1>
  struct associated_executor<executor_binder<T, Executor>, Executor1>;

// bind_executor:

template<class Executor, class T>
  executor_binder<decay_t<T>, Executor>
    bind_executor(const Executor& ex, T&& t);
template<class ExecutionContext, class T>
  executor_binder<decay_t<T>, typename ExecutionContext::executor_type>
    bind_executor(ExecutionContext& ctx, T&& t);

template<class Executor>
  class executor_work_guard;

// make_work_guard:

template<class Executor>
  executor_work_guard<Executor>
    make_work_guard(const Executor& ex);
template<class ExecutionContext>
  executor_work_guard<typename ExecutionContext::executor_type>
    make_work_guard(ExecutionContext& ctx);
template<class T>
```

```
  executor_work_guard<associated_executor_t<T>>
    make_work_guard(const T& t);
template<class T, class U>
  auto make_work_guard(const T& t, U&& u)
    -> decltype(make_work_guard(get_associated_executor(t, forward<U>(u))));

class system_executor;
class system_context;

bool operator==(const system_executor&, const system_executor&);
bool operator!=(const system_executor&, const system_executor&);

class bad_executor;

class executor;

bool operator==(const executor& a, const executor& b) noexcept;
bool operator==(const executor& e, nullptr_t) noexcept;
bool operator==(nullptr_t, const executor& e) noexcept;
bool operator!=(const executor& a, const executor& b) noexcept;
bool operator!=(const executor& e, nullptr_t) noexcept;
bool operator!=(nullptr_t, const executor& e) noexcept;

// dispatch:

template<class CompletionToken>
  DEDUCED dispatch(CompletionToken&& token);
template<class Executor, class CompletionToken>
  DEDUCED dispatch(const Executor& ex, CompletionToken&& token);
template<class ExecutionContext, class CompletionToken>
  DEDUCED dispatch(ExecutionContext& ctx, CompletionToken&& token);

// post:

template<class CompletionToken>
  DEDUCED post(CompletionToken&& token);
template<class Executor, class CompletionToken>
  DEDUCED post(const Executor& ex, CompletionToken&& token);
template<class ExecutionContext, class CompletionToken>
  DEDUCED post(ExecutionContext& ctx, CompletionToken&& token);

// defer:

template<class CompletionToken>
  DEDUCED defer(CompletionToken&& token);
template<class Executor, class CompletionToken>
  DEDUCED defer(const Executor& ex, CompletionToken&& token);
template<class ExecutionContext, class CompletionToken>
  DEDUCED defer(ExecutionContext& ctx, CompletionToken&& token);

template<class Executor>
  class strand;

template<class Executor>
  bool operator==(const strand<Executor>& a, const strand<Executor>& b);
```

```
    template<class Executor>
      bool operator!=(const strand<Executor>& a, const strand<Executor>& b);

    template<class ProtoAllocator = allocator<void>>
      class use_future_t;

    constexpr use_future_t<> use_future = use_future_t<>();

    template<class ProtoAllocator, class Result, class... Args>
      class async_result<use_future_t<ProtoAllocator>, Result(Args...)>;

    template<class R, class... Args, class Signature>
      class async_result<packaged_task<Result(Args...)>, Signature>;

  } // inline namespace v1
  } // namespace net
  } // namespace experimental

    template<class Allocator>
      struct uses_allocator<experimental::net::v1::executor, Allocator>
        : true_type {};

  } // namespace std
```

13.2 Requirements [async.reqmts]

13.2.1 Proto-allocator requirements [async.reqmts.proto.allocator]

1 A type `A` meets the proto-allocator requirements if `A` is `CopyConstructible` (C++Std [copyconstructible]), `Destructible` (C++Std [destructible]), and `allocator_traits<A>::rebind_alloc<U>` meets the allocator requirements (C++Std [allocator.requirements]), where `U` is an object type. [*Note:* For example, `std::allocator<void>` meets the proto-allocator requirements but not the allocator requirements. — *end note*] No constructor, comparison operator, copy operation, move operation, or swap operation on these types shall exit via an exception.

13.2.2 Executor requirements [async.reqmts.executor]

1 The library describes a standard set of requirements for executors. A type meeting the `Executor` requirements embodies a set of rules for determining how submitted function objects are to be executed.

2 A type `X` meets the `Executor` requirements if it satisfies the requirements of `CopyConstructible` (C++Std [copyconstructible]) and `Destructible` (C++Std [destructible]), as well as the additional requirements listed below.

3 No constructor, comparison operator, copy operation, move operation, swap operation, or member functions `context`, `on_work_started`, and `on_work_finished` on these types shall exit via an exception.

4 The executor copy constructor, comparison operators, and other member functions defined in these requirements shall not introduce data races as a result of concurrent calls to those functions from different threads.

5 Let `ctx` be the execution context returned by the executor's `context()` member function. An executor becomes invalid when the first call to `ctx.shutdown()` returns. The effect of calling `on_work_started`, `on_work_finished`, `dispatch`, `post`, or `defer` on an invalid executor is undefined. [*Note:* The copy constructor, comparison operators, and `context()` member function continue to remain valid until `ctx` is destroyed. — *end note*]

6 In Table 4, `x1` and `x2` denote (possibly const) values of type `X`, `mx1` denotes an xvalue of type `X`, `f` denotes a

`MoveConstructible` (C++Std [moveconstructible]) function object callable with zero arguments, `a` denotes a (possibly const) value of type `A` meeting the `Allocator` requirements (C++Std [allocator.requirements]), and `u` denotes an identifier.

Table 4 — Executor requirements

expression	type	assertion/note pre/post-conditions
`X u(x1);`		Shall not exit via an exception. *post:* `u == x1` and `std::addressof(u.context()) == std::addressof(x1.context())`.
`X u(mx1);`		Shall not exit via an exception. *post:* `u` equals the prior value of `mx1` and `std::addressof(u.context())` equals the prior value of `std::addressof(mx1.context())`.
`x1 == x2`	`bool`	Returns `true` only if `x1` and `x2` can be interchanged with identical effects in any of the expressions defined in these type requirements. [*Note:* Returning `false` does not necessarily imply that the effects are not identical. — *end note*] `operator==` shall be reflexive, symmetric, and transitive, and shall not exit via an exception.
`x1 != x2`	`bool`	Same as `!(x1 == x2)`.
`x1.context()`	`execution_context&`, or `E&` where `E` is a type that satisfies the `ExecutionContext` (13.2.3) requirements.	Shall not exit via an exception. The comparison operators and member functions defined in these requirements shall not alter the reference returned by this function.
`x1.on_work_started()`		Shall not exit via an exception.
`x1.on_work_finished()`		Shall not exit via an exception. *Precondition:* A preceding call `x2.on_work_started()` where `x1 == x2`.

Table 4 — Executor requirements (continued)

expression	**type**	**assertion/note pre/post-conditions**
`x1.dispatch(std::move(f), a)`		*Effects:* Creates an object `f1` initialized with *`DECAY_COPY`*`(forward<Func>(f))` (C++Std [thread.decaycopy]) in the current thread of execution . Calls `f1()` at most once. The executor may block forward progress of the caller until `f1()` finishes execution. Executor implementations should use the supplied allocator to allocate any memory required to store the function object. Prior to invoking the function object, the executor shall deallocate any memory allocated. [*Note:* Executors defined in this Technical Specification always use the supplied allocator unless otherwise specified. — *end note*] *Synchronization:* The invocation of `dispatch` synchronizes with (C++Std [intro.multithread]) the invocation of `f1`.
`x1.post(std::move(f), a)` `x1.defer(std::move(f), a)`		*Effects:* Creates an object `f1` initialized with *`DECAY_COPY`*`(forward<Func>(f))` in the current thread of execution. Calls `f1()` at most once. The executor shall not block forward progress of the caller pending completion of `f1()`. Executor implementations should use the supplied allocator to allocate any memory required to store the function object. Prior to invoking the function object, the executor shall deallocate any memory allocated. [*Note:* Executors defined in this Technical Specification always use the supplied allocator unless otherwise specified. — *end note*] *Synchronization:* The invocation of `post` or `defer` synchronizes with (C++Std [intro.multithread]) the invocation of `f1`. [*Note:* Although the requirements placed on `defer` are identical to `post`, the use of `post` conveys a preference that the caller does not block the first step of `f1`'s progress, whereas `defer` conveys a preference that the caller does block the first step of `f1`. One use of `defer` is to convey the intention of the caller that `f1` is a continuation of the current call context. The executor may use this information to optimize or otherwise adjust the way in which `f1` is invoked. — *end note*]

13.2.3 Execution context requirements [async.reqmts.executioncontext]

1 A type `X` meets the `ExecutionContext` requirements if it is publicly and unambiguously derived from `execution_context`, and satisfies the additional requirements listed below.

2 In Table 5, `x` denotes a value of type `X`.

Table 5 — ExecutionContext requirements

expression	return type	assertion/note pre/post-condition
`X::executor_type`	type meeting `Executor` (13.2.2) requirements	
`x.~X()`		Destroys all unexecuted function objects that were submitted via an executor object that is associated with the execution context.
`x.get_executor()`	`X::executor_type`	Returns an executor object that is associated with the execution context.

13.2.4 Service requirements [async.reqmts.service]

1 A class is a service if it is publicly and unambiguously derived from `execution_context::service`, or if it is publicly and unambiguously derived from another service. For a service `S`, `S::key_type` shall be valid and denote a type (C++Std [temp.deduct]), `is_base_of_v<typename S::key_type, S>` shall be `true`, and `S` shall satisfy the `Destructible` requirements (C++Std [destructible]).

2 The first parameter of all service constructors shall be an lvalue reference to `execution_context`. This parameter denotes the `execution_context` object that represents a set of services, of which the service object will be a member. [*Note:* These constructors may be called by the `make_service` function. — *end note*]

3 A service shall provide an explicit constructor with a single parameter of lvalue reference to `execution_context`. [*Note:* This constructor may be called by the `use_service` function. — *end note*]

4 [*Example:*

```
class my_service : public execution_context::service
{
public:
  using key_type = my_service;
  explicit my_service(execution_context& ctx);
  my_service(execution_context& ctx, int some_value);
private:
  virtual void shutdown() noexcept override;
  ...
};
```

— *end example*]

5 A service's `shutdown` member function shall destroy all copies of user-defined function objects that are held by the service.

13.2.5 Signature requirements [async.reqmts.signature]

1 A type satisfies the signature requirements if it is a call signature (C++Std [func.def]).

13.2.6 Associator requirements [async.reqmts.associator]

1 An associator defines a relationship between different types and objects where, given:

(1.1) — a source object `s` of type `S`,

(1.2) — type requirements `R`, and

(1.3) — a candidate object `c` of type `C` meeting the type requirements `R`,

an associated type `A` meeting the type requirements `R` may be computed, and an associated object `a` of type `A` may be obtained.

2 An associator shall be a class template that takes two template type arguments. The first template argument is the source type `S`. The second template argument is the candidate type `C`. The second template argument shall be defaulted to some default candidate type `D` that satisfies the type requirements `R`.

3 An associator shall additionally satisfy the requirements in Table 6. In this table, `X` is a class template that meets the associator requirements, `S` is the source type, `s` is a (possibly const) value of type `S`, `C` is the candidate type, `c` is a (possibly const) value of type `C`, `D` is the default candidate type, and `d` is a (possibly const) value of type `D` that is the default candidate object.

Table 6 — Associator requirements

expression	return type	assertion/note pre/post-conditions
X<S>::type	X<S, D>::type	
X<S, C>::type		The associated type.
X<S>::get(s)	X<S>::type	Returns X<S>::get(S, d).
X<S, C>::get(s, c)	X<S, C>::type	Returns the associated object.

4 The associator's primary template shall be defined. A program may partially specialize the associator class template for some user-defined type `S`.

5 Finally, the associator shall provide the following type alias and function template in the enclosing namespace:

```
template<class S, class C = D> using X_t = typename X<S, C>::type;

template<class S, class C = D>
typename X<S, C>::type get_X(const S& s, const C& c = d)
{
  return X<S, C>::get(s, c);
}
```

where X is replaced with the name of the associator class template. [*Note:* This function template is provided as a convenience, to automatically deduce the source and candidate types. *— end note*]

13.2.7 Requirements on asynchronous operations [async.reqmts.async]

1 This section uses the names `Alloc1`, `Alloc2`, `alloc1`, `alloc2`, `Args`, `CompletionHandler`, `completion_handler`, `Executor1`, `Executor2`, `ex1`, `ex2`, `f`, *i*, *N*, `Signature`, `token`, $\mathtt{T}_i$, $\mathtt{t}_i$, `work1`, and `work2` as placeholders for specifying the requirements below.

13.2.7.1 General asynchronous operation concepts [async.reqmts.async.concepts]

1 An initiating function is a function which may be called to start an asynchronous operation. A completion handler is a function object that will be invoked, at most once, with the result of the asynchronous operation.

2 The life cycle of an asynchronous operation is comprised of the following events and phases:

(2.1) — Event 1: The asynchronous operation is started by a call to the initiating function.

(2.2) — Phase 1: The asynchronous operation is now outstanding.

(2.3) — Event 2: The externally observable side effects of the asynchronous operation, if any, are fully established. The completion handler is submitted to an executor.

(2.4) — Phase 2: The asynchronous operation is now completed.

(2.5) — Event 3: The completion handler is called with the result of the asynchronous operation.

3 In this Technical Specification, all functions with the prefix `async_` are initiating functions.

13.2.7.2 Completion tokens and handlers [async.reqmts.async.token]

1 Initiating functions:

(1.1) — are function templates with template parameter `CompletionToken`;

(1.2) — accept, as the final parameter, a completion token object `token` of type `CompletionToken`;

(1.3) — specify a completion signature, which is a call signature (C++Std [func.def]) `Signature` that determines the arguments to the completion handler.

2 An initiating function determines the type `CompletionHandler` of its completion handler function object by performing `typename async_result<decay_t<CompletionToken>, Signature>::completion_handler_type`. The completion handler object `completion_handler` is initialized with `std::forward<CompletionToken>(token)`. [*Note:* No other requirements are placed on the type `CompletionToken`. — *end note*]

3 The type `CompletionHandler` must satisfy the requirements of `Destructible` (C++Std [destructible]) and `MoveConstructible` (C++Std [moveconstructible]), and be callable with the specified call signature.

4 In this Technical Specification, all initiating functions specify a *Completion signature:* element that defines the call signature `Signature`. The *Completion signature:* elements in this Technical Specification have named parameters, and the results of an asynchronous operation are specified in terms of these names.

13.2.7.3 Deduction of initiating function return type [async.reqmts.async.return.type]

1 The return type of an initiating function is `typename async_result<decay_t<CompletionToken>, Signature>::return_type`.

2 For the sake of exposition, this Technical Specification sometimes annotates functions with a return type *`DEDUCED`*. For every function declaration that returns *`DEDUCED`*, the meaning is equivalent to specifying the return type as `typename async_result<decay_t<CompletionToken>, Signature>::return_type`.

13.2.7.4 Production of initiating function return value [async.reqmts.async.return.value]

1 An initiating function produces its return type as follows:

(1.1) — constructing an object `result` of type `async_result<decay_t<CompletionToken>, Signature>`, initialized as `result(completion_handler)`; and

(1.2) — using `result.get()` as the operand of the return statement.

2 [*Example:* Given an asynchronous operation with Completion signature `void(R1 r1, R2 r2)`, an initiating function meeting these requirements may be implemented as follows:

```
template<class CompletionToken>
auto async_xyz(T1 t1, T2 t2, CompletionToken&& token)
{
  typename async_result<decay_t<CompletionToken>, void(R1, R2)>::completion_handler_type
    completion_handler(forward<CompletionToken>(token));

  async_result<decay_t<CompletionToken>, void(R1, R2)> result(completion_handler);
```

```
    // initiate the operation and cause completion_handler to be invoked with
    // the result

    return result.get();
  }
```

3 For convenience, initiating functions may be implemented using the `async_completion` template:

```
  template<class CompletionToken>
  auto async_xyz(T1 t1, T2 t2, CompletionToken&& token)
  {
    async_completion<CompletionToken, void(R1, R2)> init(token);

    // initiate the operation and cause init.completion_handler to be invoked
    // with the result

    return init.result.get();
  }
```

— *end example*]

13.2.7.5 Lifetime of initiating function arguments [async.reqmts.async.lifetime]

1 Unless otherwise specified, the lifetime of arguments to initiating functions shall be treated as follows:

(1.1) — If the parameter has a pointer type or has a type of lvalue reference to non-const, the implementation may assume the validity of the pointee or referent, respectively, until the completion handler is invoked. [*Note:* In other words, the program must guarantee the validity of the argument until the completion handler is invoked. — *end note*]

(1.2) — Otherwise, the implementation must not assume the validity of the argument after the initiating function completes. [*Note:* In other words, the program is not required to guarantee the validity of the argument after the initiating function completes. — *end note*] The implementation may make copies of the argument, and all copies shall be destroyed no later than immediately after invocation of the completion handler.

13.2.7.6 Non-blocking requirements on initiating functions[async.reqmts.async.non.blocking]

1 An initiating function shall not block (C++Std [defns.block]) the calling thread pending completion of the outstanding operation.

2 [*Note:* Initiating functions may still block the calling thread for other reasons. For example, an initiating function may lock a mutex in order to synchronize access to shared data. — *end note*]

13.2.7.7 Associated executor [async.reqmts.async.assoc.exec]

1 Certain objects that participate in asynchronous operations have an associated executor. These are obtained as specified below.

13.2.7.8 I/O executor [async.reqmts.async.io.exec]

1 An asynchronous operation has an associated executor satisfying the `Executor` (13.2.2) requirements. If not otherwise specified by the asynchronous operation, this associated executor is an object of type `system_executor`.

2 All asynchronous operations in this Technical Specification have an associated executor object that is determined as follows:

(2.1) — If the initiating function is a member function, the associated executor is that returned by the `get_executor` member function on the same object.

(2.2) — If the initiating function is not a member function, the associated executor is that returned by the `get_executor` member function of the first argument to the initiating function.

3 Let `Executor1` be the type of the associated executor. Let `ex1` be a value of type `Executor1`, representing the associated executor object obtained as described above.

13.2.7.9 Completion handler executor [async.reqmts.async.handler.exec]

1 A completion handler object of type `CompletionHandler` has an associated executor satisfying the Executor requirements (13.2.2). The type of this associated executor is `associated_executor_t<CompletionHandler, Executor1>`. Let `Executor2` be the type `associated_executor_t<CompletionHandler, Executor1>`. Let `ex2` be a value of type `Executor2` obtained by performing `get_associated_executor(completion_handler, ex1)`.

13.2.7.10 Outstanding work [async.reqmts.async.work]

1 Until the asynchronous operation has completed, the asynchronous operation shall maintain:

(1.1) — an object `work1` of type `executor_work_guard<Executor1>`, initialized as `work1(ex1)`, and where `work1.owns_work() == true`; and

(1.2) — an object `work2` of type `executor_work_guard<Executor2>`, initialized as `work2(ex2)`, and where `work2.owns_work() == true`.

13.2.7.11 Allocation of intermediate storage [async.reqmts.async.alloc]

1 Asynchronous operations may allocate memory. [*Note:* Such as a data structure to store copies of the `completion_handler` object and the initiating function's arguments. — *end note*]

2 Let `Alloc1` be a type, satisfying the `ProtoAllocator` (13.2.1) requirements, that represents the asynchronous operation's default allocation strategy. [*Note:* Typically `std::allocator<void>`. — *end note*] Let `alloc1` be a value of type `Alloc1`.

3 A completion handler object of type `CompletionHandler` has an associated allocator object `alloc2` of type `Alloc2` satisfying the `ProtoAllocator` (13.2.1) requirements. The type `Alloc2` is `associated_allocator_t<CompletionHandler, Alloc1>`. Let `alloc2` be a value of type `Alloc2` obtained by performing `get_associated_allocator(completion_handler, alloc1)`.

4 The asynchronous operations defined in this Technical Specification:

(4.1) — If required, allocate memory using only the completion handler's associated allocator.

(4.2) — Prior to completion handler execution, deallocate any memory allocated using the completion handler's associated allocator.

5 [*Note:* The implementation may perform operating system or underlying API calls that perform memory allocations not using the associated allocator. Invocations of the allocator functions may not introduce data races (See C++Std [res.on.data.races]). — *end note*]

13.2.7.12 Execution of completion handler on completion of asynchronous operation [async.reqmts.async.completion]

1 Let `Args...` be the argument types of the completion signature `Signature` and let N be `sizeof...(Args)`. Let i be in the range `[0,` N`)`. Let $\mathtt{T}_i$ be the i^{th} type in `Args...` and let $\mathtt{t}_i$ be the i^{th} completion handler argument associated with $\mathtt{T}_i$.

2 Let `f` be a function object, callable as `f()`, that invokes `completion_handler` as if by $\mathtt{completion\_handler(forward<T_0>(t_0),\ ...,\ forward<T_{N-1}>(t_{N-1}))}$.

3 If an asynchronous operation completes immediately (that is, within the thread of execution calling the initiating function, and before the initiating function returns), the completion handler shall be submitted for

execution as if by performing `ex2.post(std::move(f), alloc2)`. Otherwise, the completion handler shall be submitted for execution as if by performing `ex2.dispatch(std::move(f), alloc2)`.

13.2.7.13 Completion handlers and exceptions [async.reqmts.async.exceptions]

1 Completion handlers are permitted to throw exceptions. The effect of any exception propagated from the execution of a completion handler is determined by the executor which is executing the completion handler.

13.2.7.14 Composed asynchronous operations [async.reqmts.async.composed]

1 In this Technical Specification, a *composed asynchronous operation* is an asynchronous operation that is implemented in terms of zero or more intermediate calls to other asynchronous operations. The intermediate asynchronous operations are performed sequentially. [*Note:* That is, the completion handler of an intermediate operation initiates the next operation in the sequence. — *end note*]

An intermediate operation's completion handler shall have an associated executor that is either:

(1.1) — the type `Executor2` and object `ex2` obtained from the completion handler type `CompletionHandler` and object `completion_handler`; or

(1.2) — an object of an unspecified type satisfying the Executor requirements (13.2.2), that delegates executor operations to the type `Executor2` and object `ex2`.

An intermediate operation's completion handler shall have an associated allocator that is either:

(1.3) — the type `Alloc2` and object `alloc2` obtained from the completion handler type `CompletionHandler` and object `completion_handler`; or

(1.4) — an object of an unspecified type satisfying the ProtoAllocator requirements (13.2.1), that delegates allocator operations to the type `Alloc2` and object `alloc2`.

13.3 Class template async_result [async.async.result]

1 The `async_result` class template is a customization point for asynchronous operations. Template parameter `CompletionToken` specifies the model used to obtain the result of the asynchronous operation. Template parameter `Signature` is the call signature (C++Std [func.def]) for the completion handler type invoked on completion of the asynchronous operation. The `async_result` template:

(1.1) — transforms a `CompletionToken` into a completion handler type that is based on a `Signature`; and

(1.2) — determines the return type and return value of an asynchronous operation's initiating function.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class CompletionToken, class Signature>
  class async_result
  {
  public:
    using completion_handler_type = CompletionToken;
    using return_type = void;

    explicit async_result(completion_handler_type&) {}
    async_result(const async_result&) = delete;
    async_result& operator=(const async_result&) = delete;

    return_type get() {}
```

```
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The template parameter `CompletionToken` shall be an object type. The template parameter `Signature` shall be a call signature (C++Std [func.def]).

3 Specializations of `async_result` shall satisfy the `Destructible` requirements (C++Std [destructible]) in addition to the requirements in Table 7. In this table, `R` is a specialization of `async_result`; `r` is a modifiable lvalue of type `R`; and `h` is a modifiable lvalue of type `R::completion_handler_type`.

Table 7 — `async_result` specialization requirements

Expression	Return type	Requirement
`R::completion_handler_type`		A type satisfying `MoveConstructible` requirements (C++Std [moveconstructible]), An object of type `completion_handler_type` shall be a function object with call signature `Signature`, and `completion_handler_type` shall be constructible with an rvalue of type `CompletionToken`.
`R::return_type`		`void`; or a type satisfying `MoveConstructible` requirements (C++Std [moveconstructible])
`R r(h);`		
`r.get()`	`R::return_type`	[*Note:* An asynchronous operation's initiating function uses the `get()` member function as the sole operand of a return statement. — *end note*]

13.4 Class template `async_completion` [async.async.completion]

1 Class template `async_completion` is provided as a convenience, to simplify the implementation of asynchronous operations that use `async_result`.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class CompletionToken, class Signature>
  struct async_completion
  {
    using completion_handler_type = async_result<decay_t<CompletionToken>,
      Signature>::completion_handler_type;

    explicit async_completion(CompletionToken& t);
    async_completion(const async_completion&) = delete;
    async_completion& operator=(const async_completion&) = delete;

    see below completion_handler;
    async_result<decay_t<CompletionToken>, Signature> result;
```

```
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The template parameter `Signature` shall be a call signature (C++Std [func.def]).

`explicit async_completion(CompletionToken& t);`

3 *Effects:* If `CompletionToken` and `completion_handler_type` are the same type, binds `completion_handler` to `t`; otherwise, initializes `completion_handler` with `forward<CompletionToken>(t)`. Initializes `result` with `completion_handler`.

see below `completion_handler;`

4 *Type:* `completion_handler_type&` if `CompletionToken` and `completion_handler_type` are the same type; otherwise, `completion_handler_type`.

13.5 Class template `associated_allocator` [async.assoc.alloc]

1 Class template `associated_allocator` is an associator (13.2.6) for the `ProtoAllocator` (13.2.1) type requirements, with default candidate type `allocator<void>` and default candidate object `allocator<void>()`.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class ProtoAllocator = allocator<void>>
  struct associated_allocator
  {
    using type = see below;

    static type get(const T& t, const ProtoAllocator& a = ProtoAllocator()) noexcept;
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Specializations of `associated_allocator` shall satisfy the requirements in Table 8. In this table, `X` is a specialization of `associated_allocator` for the template parameters `T` and `ProtoAllocator`; `t` is a value of (possibly const) `T`; and `a` is an object of type `ProtoAllocator`.

Table 8 — `associated_allocator` specialization requirements

Expression	Return type	Note
`typename X::type`	A type meeting the proto-allocator (13.2.1) requirements.	
`X::get(t)`	`X::type`	Shall not exit via an exception. Equivalent to `X::get(t, ProtoAllocator())`.
`X::get(t, a)`	`X::type`	Shall not exit via an exception.

13.5.1 `associated_allocator` members [async.assoc.alloc.members]

`using type = see below;`

1 *Type:* If the *qualified-id* `T::allocator_type` is valid and denotes a type (C++Std [temp.deduct]), `T::allocator_type`. Otherwise `ProtoAllocator`.

`type get(const T& t, const ProtoAllocator& a = ProtoAllocator()) noexcept;`

2 *Returns:* If the *qualified-id* `T::allocator_type` is valid and denotes a type (C++Std [temp.deduct]), `t.get_allocator()`. Otherwise `a`.

13.6 Function `get_associated_allocator` [async.assoc.alloc.get]

```
template<class T>
  associated_allocator_t<T> get_associated_allocator(const T& t) noexcept;
```

1 *Returns:* `associated_allocator<T>::get(t)`.

```
template<class T, class ProtoAllocator>
  associated_allocator_t<T, ProtoAllocator>
    get_associated_allocator(const T& t, const ProtoAllocator& a) noexcept;
```

2 *Returns:* `associated_allocator<T, ProtoAllocator>::get(t, a)`.

13.7 Class `execution_context` [async.exec.ctx]

1 Class `execution_context` implements an extensible, type-safe, polymorphic set of services, indexed by service type.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class execution_context
  {
  public:
    class service;

    // construct / copy / destroy:

    execution_context();
    execution_context(const execution_context&) = delete;
    execution_context& operator=(const execution_context&) = delete;
    virtual ~execution_context();

    // execution context operations:

    void notify_fork(fork_event e);

  protected:

    // execution context protected operations:

    void shutdown() noexcept;
    void destroy() noexcept;
  };
```

```
  // service access:
  template<class Service> typename Service::key_type&
    use_service(execution_context& ctx);
  template<class Service, class... Args> Service&
    make_service(execution_context& ctx, Args&&... args);
  template<class Service> bool has_service(const execution_context& ctx) noexcept;
  class service_already_exists : public logic_error { };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Access to the services of an `execution_context` is via three function templates, `use_service<>`, `make_service<>` and `has_service<>`.

3 In a call to `use_service<Service>()`, the type argument chooses a service. If the service is not present in an `execution_context`, an object of type `Service` is created and added to the `execution_context`. A program can check if an `execution_context` implements a particular service with the function template `has_service<Service>()`.

4 Service objects may be explicitly added to an `execution_context` using the function template `make_service<Service>()`. If the service is already present, `make_service` exits via an exception of type `service_already_exists`.

5 Once a service reference is obtained from an `execution_context` object by calling `use_service<>`, that reference remains usable until a call to `destroy()`.

13.7.1 `execution_context` constructor [async.exec.ctx.cons]

`execution_context();`

1 *Effects:* Creates an object of class `execution_context` which contains no services. [*Note:* An implementation might preload services of internal service types for its own use. — *end note*]

13.7.2 `execution_context` destructor [async.exec.ctx.dtor]

`~execution_context();`

1 *Effects:* Destroys an object of class `execution_context`. Performs `shutdown()` followed by `destroy()`.

13.7.3 `execution_context` operations [async.exec.ctx.ops]

`void notify_fork(fork_event e);`

1 *Effects:* For each service object `svc` in the set:

(1.1) — If `e == fork_event::prepare`, performs `svc->notify_fork(e)` in reverse order of addition to the set.

(1.2) — Otherwise, performs `svc->notify_fork(e)` in order of addition to the set.

13.7.4 `execution_context` protected operations [async.exec.ctx.protected]

`void shutdown() noexcept;`

1 *Effects:* For each service object `svc` in the `execution_context` set, in reverse order of addition to the set, performs `svc->shutdown()`. For each service in the set, `svc->shutdown()` is called only once irrespective of the number of calls to `shutdown` on the `execution_context`.

```
void destroy() noexcept;
```

2 *Effects:* Destroys each service object in the `execution_context` set, and removes it from the set, in reverse order of addition to the set.

13.7.5 `execution_context` globals [async.exec.ctx.globals]

1 The functions `use_service`, `make_service`, and `has_service` do not introduce data races as a result of concurrent calls to those functions from different threads.

```
template<class Service> typename Service::key_type&
  use_service(execution_context& ctx);
```

2 *Effects:* If an object of type `Service::key_type` does not already exist in the `execution_context` set identified by `ctx`, creates an object of type `Service`, initialized as `Service(ctx)`, and adds it to the set.

3 *Returns:* A reference to the corresponding service of `ctx`.

4 Notes: The reference returned remains valid until a call to `destroy`.

```
template<class Service, class... Args> Service&
  make_service(execution_context& ctx, Args&&... args);
```

5 *Requires:* A service object of type `Service::key_type` does not already exist in the `execution_context` set identified by `ctx`.

6 *Effects:* Creates an object of type `Service`, initialized as `Service(ctx, forward<Args>(args)...)`, and adds it to the `execution_context` set identified by `ctx`.

7 *Remarks:* `service_already_exists` if a corresponding service object of type `Key` is already present in the set.

8 Notes: The reference returned remains valid until a call to `destroy`.

```
template<class Service> bool has_service(const execution_context& ctx) noexcept;
```

9 *Returns:* `true` if an object of type `Service::key_type` is present in `ctx`, otherwise `false`.

13.8 Class `execution_context::service` [async.exec.ctx.svc]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class execution_context::service
  {
  protected:
    // construct / copy / destroy:

    explicit service(execution_context& owner);
    service(const service&) = delete;
    service& operator=(const service&) = delete;
    virtual ~service();

    // service observers:

    execution_context& context() noexcept;
```

```
  private:
    // service operations:

    virtual void shutdown() noexcept = 0;
    virtual void notify_fork(fork_event e) {}

    execution_context& context_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

```
explicit service(execution_context& owner);
```

1 *Postconditions:* `std::addressof(context_) == std::addressof(owner)`.

```
execution_context& context() noexcept;
```

2 *Returns:* `context_`.

13.9 Class template `is_executor` [async.is.exec]

1 The class template `is_executor` can be used to detect executor types satisfying the `Executor` (13.2.2) type requirements.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T> struct is_executor;

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 `T` shall be a complete type.

3 Class template `is_executor` is a UnaryTypeTrait (C++Std [meta.rqmts]) with a BaseCharacteristic of `true_type` if the type `T` meets the syntactic requirements for `Executor` (13.2.2), otherwise `false_type`.

13.10 Executor argument tag [async.executor.arg]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  struct executor_arg_t { };
  constexpr executor_arg_t executor_arg = executor_arg_t();

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 The `executor_arg_t` struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, types may have constructors with `executor_arg_t` as the first argument, immediately followed by an argument of a type that satisfies the Executor requirements (13.2.2).

13.11 `uses_executor` [async.uses.executor]

13.11.1 `uses_executor` trait [async.uses.executor.trait]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor> struct uses_executor;

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 Remark: Detects whether `T` has a nested `executor_type` that is convertible from `Executor`. Meets the `BinaryTypeTrait` requirements (C++Std [meta.rqmts]). The implementation provides a definition that is derived from `true_type` if a type `T::executor_type` exists and `is_convertible<Executor, T::executor_type>::value != false`, otherwise it is derived from `false_type`. A program may specialize this template to derive from `true_type` for a user-defined type `T` that does not have a nested `executor_type` but nonetheless can be constructed with an executor if the first argument of a constructor has type `executor_arg_t` and the second argument has type `Executor`.

13.11.2 uses-executor construction [async.uses.executor.cons]

1 Uses-executor construction with executor `Executor` refers to the construction of an object `obj` of type `T`, using constructor arguments `v1, v2, ..., vN` of types `V1, V2, ..., VN`, respectively, and an executor `ex` of type `Executor`, according to the following rules:

(1.1) — if `uses_executor_v<T, Executor>` is `true` and `is_constructible<T, executor_arg_t, Executor, V1, V2, ..., VN>::value` is `true`, then `obj` is initialized as `obj(executor_arg, ex, v1, v2, ..., vN)`;

(1.2) — otherwise, `obj` is initialized as `obj(v1, v2, ..., vN)`.

13.12 Class template `associated_executor` [async.assoc.exec]

1 Class template `associated_executor` is an associator (13.2.6) for the `Executor` (13.2.2) type requirements, with default candidate type `system_executor` and default candidate object `system_executor()`.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor = system_executor>
  struct associated_executor
  {
    using type = see below;

    static type get(const T& t, const Executor& e = Executor()) noexcept;
  };
```

```
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Specializations of `associated_executor` shall satisfy the requirements in Table 9. In this table, `X` is a specialization of `associated_executor` for the template parameters `T` and `Executor`; `t` is a value of (possibly const) `T`; and `e` is an object of type `Executor`.

Table 9 — `associated_executor` specialization requirements

Expression	Return type	Note
`typename X::type`	A type meeting Executor requirements (13.2.2).	
`X::get(t)`	`X::type`	Shall not exit via an exception. Equivalent to `X::get(t, Executor())`.
`X::get(t, e)`	`X::type`	Shall not exit via an exception.

13.12.1 `associated_executor` members [async.assoc.exec.members]

`using type = see below;`

1 *Type:* If the *qualified-id* `T::executor_type` is valid and denotes a type (C++Std [temp.deduct]), `T::executor_type`. Otherwise `Executor`.

`type get(const T& t, const Executor& e = Executor()) noexcept;`

2 *Returns:* If the *qualified-id* `T::executor_type` is valid and denotes a type (C++Std [temp.deduct]), `t.get_executor()`. Otherwise `e`.

13.13 Function `get_associated_executor` [async.assoc.exec.get]

```
template<class T>
  associated_executor_t<T> get_associated_executor(const T& t) noexcept;
```

1 *Returns:* `associated_executor<T>::get(t)`.

```
template<class T, class Executor>
  associated_executor_t<T, Executor>
    get_associated_executor(const T& t, const Executor& ex) noexcept;
```

2 *Returns:* `associated_executor<T, Executor>::get(t, ex)`.

3 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<Executor>` is `true`.

```
template<class T, class ExecutionContext>
  associated_executor_t<T, typename ExecutionContext::executor_type>
    get_associated_executor(const T& t, ExecutionContext& ctx) noexcept;
```

4 *Returns:* `get_associated_executor(t, ctx.get_executor())`.

5 *Remarks:* This function shall not participate in overload resolution unless `is_convertible<ExecutionContext&, execution_context&>::value` is `true`.

13.14 Class template `executor_binder` [async.exec.binder]

1 The class template `executor_binder` binds executors to objects. A specialization `executor_binder<T, Executor>` binds an executor of type `Executor` satisfying the Executor requirements (13.2.2) to an object or function of type `T`.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor>
  class executor_binder
  {
  public:
    // types:

    using target_type = T;
    using executor_type = Executor;

    // construct / copy / destroy:

    executor_binder(T t, const Executor& ex);
    executor_binder(const executor_binder& other) = default;
    executor_binder(executor_binder&& other) = default;
    template<class U, class OtherExecutor>
      executor_binder(const executor_binder<U, OtherExecutor>& other);
    template<class U, class OtherExecutor>
      executor_binder(executor_binder<U, OtherExecutor>&& other);
    template<class U, class OtherExecutor>
      executor_binder(executor_arg_t, const Executor& ex,
        const executor_binder<U, OtherExecutor>& other);
    template<class U, class OtherExecutor>
      executor_binder(executor_arg_t, const Executor& ex,
        executor_binder<U, OtherExecutor>&& other);

    ~executor_binder();

    // executor binder access:

    T& get() noexcept;
    const T& get() const noexcept;
    executor_type get_executor() const noexcept;

    // executor binder invocation:

    template<class... Args>
      result_of_t<T&(Args&&...)> operator()(Args&&... args);
    template<class... Args>
      result_of_t<const T&(Args&&...)> operator()(Args&&... args) const;

  private:
    Executor ex_; // exposition only
    T target_; // exposition only
  };
```

```
    template<class T, class Executor, class Signature>
      class async_result<executor_binder<T, Executor>, Signature>;

    template<class T, class Executor, class ProtoAllocator>
      struct associated_allocator<executor_binder<T, Executor>, ProtoAllocator>;

    template<class T, class Executor, class Executor1>
      struct associated_executor<executor_binder<T, Executor>, Executor1>;

  } // inline namespace v1
  } // namespace net
  } // namespace experimental
  } // namespace std
```

13.14.1 executor_binder constructors [async.exec.binder.cons]

```
executor_binder(T t, const Executor& ex);
```

1 *Effects:* Initializes `ex_` with `ex`. Initializes `target_` by performing uses-executor construction, using the constructor argument `std::move(t)` and the executor `ex_`.

```
template<class U, class OtherExecutor>
  executor_binder(const executor_binder<U, OtherExecutor>& other);
```

2 *Requires:* If `U` is not convertible to `T`, or if `OtherExecutor` is not convertible to `Executor`, the program is ill-formed.

3 *Effects:* Initializes `ex_` with `other.get_executor()`. Initializes `target_` by performing uses-executor construction, using the constructor argument `other.get()` and the executor `ex_`.

```
template<class U, class OtherExecutor>
  executor_binder(executor_binder<U, OtherExecutor>&& other);
```

4 *Requires:* If `U` is not convertible to `T`, or if `OtherExecutor` is not convertible to `Executor`, the program is ill-formed.

5 *Effects:* Initializes `ex_` with `other.get_executor()`. Initializes `target_` by performing uses-executor construction, using the constructor argument `std::move(other.get())` and the executor `ex_`.

```
template<class U, class OtherExecutor>
  executor_binder(executor_arg_t, const Executor& ex,
    const executor_binder<U, OtherExecutor>& other);
```

6 *Requires:* If `U` is not convertible to `T` the program is ill-formed.

7 *Effects:* Initializes `ex_` with `ex`. Initializes `target_` by performing uses-executor construction, using the constructor argument `other.get()` and the executor `ex_`.

```
template<class U, class OtherExecutor>
  executor_binder(executor_arg_t, const Executor& ex,
    executor_binder<U, OtherExecutor>&& other);
```

8 *Requires:* `U` is `T` or convertible to `T`.

9 *Effects:* Initializes `ex_` with `ex`. Initializes `target_` by performing uses-executor construction, using the constructor argument `std::move(other.get())` and the executor `ex_`.

13.14.2 executor_binder access [async.exec.binder.access]

```
T& get() noexcept;
const T& get() const noexcept;
```

1 *Returns:* `target_`.

```
executor_type get_executor() const noexcept;
```

2 *Returns:* `executor_`.

13.14.3 executor_binder invocation [async.exec.binder.invocation]

```
template<class... Args>
  result_of_t<T&(Args&&...)> operator()(Args&&... args);
template<class... Args>
  result_of_t<const T&(Args&&...)> operator()(Args&&... args) const;
```

1 *Returns:* *INVOKE*`(get(), forward<Args>(args)...)` (C++Std [func.require]).

13.14.4 Class template partial specialization async_result [async.exec.binder.async.result]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor, class Signature>
  class async_result<executor_binder<T, Executor>, Signature>
  {
  public:
    using completion_handler_type = executor_binder<
      typename async_result<T, Signature>::completion_handler_type,
        Executor>;
    using return_type = typename async_result<T, Signature>::return_type;

    explicit async_result(completion_handler_type& h);
    async_result(const async_result&) = delete;
    async_result& operator=(const async_result&) = delete;

    return_type get();

  private:
    async_result<T, Signature> target_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

```
explicit async_result(completion_handler_type& h);
```

1 *Effects:* Initializes `target_` as `target_(h.get())`.

```
return_type get();
```

2 *Returns:* `target_.get()`.

13.14.5 Class template partial specialization associated_allocator [async.exec.binder.assoc.alloc]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor, class ProtoAllocator>
    struct associated_allocator<executor_binder<T, Executor>, ProtoAllocator>
  {
    using type = associated_allocator_t<T, ProtoAllocator>;

    static type get(const executor_binder<T, Executor>& b,
                    const ProtoAllocator& a = ProtoAllocator()) noexcept;
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

```
static type get(const executor_binder<T, Executor>& b,
                const ProtoAllocator& a = ProtoAllocator()) noexcept;
```

1 *Returns:* `associated_allocator<T, ProtoAllocator>::get(b.get(), a)`.

13.14.6 Class template partial specialization associated_executor [async.exec.binder.assoc.exec]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T, class Executor, class Executor1>
    struct associated_executor<executor_binder<T, Executor>, Executor1>
  {
    using type = Executor;

    static type get(const executor_binder<T, Executor>& b,
                    const Executor1& e = Executor1()) noexcept;
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

```
static type get(const executor_binder<T, Executor>& b,
                const Executor1& e = Executor1()) noexcept;
```

1 *Returns:* `b.get_executor()`.

13.15 Function bind_executor [async.bind.executor]

```
template<class Executor, class T>
```

```
  executor_binder<decay_t<T>, Executor>
    bind_executor(const Executor& ex, T&& t);
```

1 *Returns:* `executor_binder<decay_t<T>, Executor>(forward<T>(t), ex)`.

2 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<Executor>` is `true`.

```
template<class ExecutionContext, class CompletionToken>
  executor_binder<decay_t<T>, typename ExecutionContext::executor_type>
    bind_executor(ExecutionContext& ctx, T&& t);
```

3 *Returns:* `bind_executor(ctx.get_executor(), forward<T>(t))`.

4 *Remarks:* This function shall not participate in overload resolution unless `is_convertible<ExecutionContext&, execution_context&>::value` is `true`.

13.16 Class template `executor_work_guard` [async.exec.work.guard]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Executor>
  class executor_work_guard
  {
  public:
    // types:

    using executor_type = Executor;

    // construct / copy / destroy:

    explicit executor_work_guard(const executor_type& ex) noexcept;
    executor_work_guard(const executor_work_guard& other) noexcept;
    executor_work_guard(executor_work_guard&& other) noexcept;

    executor_work_guard& operator=(const executor_work_guard&) = delete;

    ~executor_work_guard();

    // executor work guard observers:

    executor_type get_executor() const noexcept;
    bool owns_work() const noexcept;

    // executor work guard modifiers:

    void reset() noexcept;

  private:
    Executor ex_; // exposition only
    bool owns_; // exposition only
  };

} // inline namespace v1
} // namespace net
```

```
} // namespace experimental
} // namespace std
```

13.16.1 `executor_work_guard` members [async.exec.work.guard.members]

```
explicit executor_work_guard(const executor_type& ex) noexcept;
```

1 *Effects:* Initializes `ex_` with `ex`, and then performs `ex_.on_work_started()`.

2 *Postconditions:* `ex_ == ex` and `owns_ == true`.

```
executor_work_guard(const executor_work_guard& other) noexcept;
```

3 *Effects:* Initializes `ex_` with `other.ex_`. If `other.owns_ == true`, performs `ex_.on_work_started()`.

4 *Postconditions:* `ex_ == other.ex_` and `owns_ == other.owns_`.

```
executor_work_guard(executor_work_guard&& other) noexcept;
```

5 *Effects:* Initializes `ex_` with `std::move(other.ex_)` and initializes `owns_` with `other.owns_`, and sets `other.owns_` to `false`.

```
~executor_work_guard();
```

6 *Effects:* If `owns_` is `true`, performs `ex_.on_work_finished()`.

```
executor_type get_executor() const noexcept;
```

7 *Returns:* `ex_`.

```
bool owns_work() const noexcept;
```

8 *Returns:* `owns_`.

```
void reset() noexcept;
```

9 *Effects:* If `owns_` is `true`, performs `ex_.on_work_finished()`.

10 *Postconditions:* `owns_ == false`.

13.17 Function `make_work_guard` [async.make.work.guard]

```
template<class Executor>
  executor_work_guard<Executor>
    make_work_guard(const Executor& ex);
```

1 *Returns:* `executor_work_guard<Executor>(ex)`.

2 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<Executor>` is `true`.

```
template<class ExecutionContext>
  executor_work_guard<typename ExecutionContext::executor_type>
    make_work_guard(ExecutionContext& ctx);
```

3 *Returns:* `make_work_guard(ctx.get_executor())`.

4 *Remarks:* This function shall not participate in overload resolution unless `is_convertible<ExecutionContext&, execution_context&>::value` is `true`.

```
template<class T>
  executor_work_guard<associated_executor_t<T>>
    make_work_guard(const T& t);
```

5 *Returns:* `make_work_guard(get_associated_executor(t))`.

6 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<T>` is `false` and `is_convertible<T&, execution_context&>::value` is `false`.

```
template<class T, class U>
  auto make_work_guard(const T& t, U&& u)
    -> decltype(make_work_guard(get_associated_executor(t, forward<U>(u))));
```

7 *Returns:* `make_work_guard(get_associated_executor(t, forward<U>(u)))`.

13.18 Class `system_executor` [async.system.exec]

1 Class `system_executor` represents a set of rules where function objects are permitted to execute on any thread.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class system_executor
  {
  public:
    // constructors:

    system_executor() {}

    // executor operations:

    system_context& context() const noexcept;

    void on_work_started() const noexcept {}
    void on_work_finished() const noexcept {}

    template<class Func, class ProtoAllocator>
      void dispatch(Func&& f, const ProtoAllocator& a) const;
    template<class Func, class ProtoAllocator>
      void post(Func&& f, const ProtoAllocator& a) const;
    template<class Func, class ProtoAllocator>
      void defer(Func&& f, const ProtoAllocator& a) const;
  };

  bool operator==(const system_executor&, const system_executor&) noexcept;
  bool operator!=(const system_executor&, const system_executor&) noexcept;

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Class `system_executor` satisfies the `Destructible` (C++Std [destructible]), `DefaultConstructible` (C++Std [defaultconstructible]), and `Executor` (13.2.2) type requirements.

3 To satisfy the `Executor` requirements for the `post` and `defer` member functions, the system executor may create `thread` objects to run the submitted function objects. These `thread` objects are collectively referred to as system threads.

13.18.1 `system_executor` operations [async.system.exec.ops]

```
system_context& context() const noexcept;
```

1 *Returns:* A reference to an object with static storage duration. All calls to this function return references to the same object.

```
template<class Func, class ProtoAllocator>
  void dispatch(Func&& f, const ProtoAllocator& a) const;
```

2 *Effects:* Equivalent to *`DECAY_COPY`*`(forward<Func>(f))()` (C++Std [thread.decaycopy]).

```
template<class Func, class ProtoAllocator>
  void post(Func&& f, const ProtoAllocator& a) const;
template<class Func, class ProtoAllocator>
  void defer(Func&& f, const ProtoAllocator& a) const;
```

3 *Effects:* If `context().stopped() == false`, creates an object `f1` initialized with *`DECAY_COPY`*`(forward<Func>(f))`, and calls `f1` as if in a thread of execution represented by a `thread` object. Any exception propagated from the execution of *`DECAY_COPY`*`(forward<Func>(f))()` results in a call to `std::terminate`.

13.18.2 `system_executor` comparisons [async.system.exec.comparisons]

```
bool operator==(const system_executor&, const system_executor&) noexcept;
```

1 *Returns:* `true`.

```
bool operator!=(const system_executor&, const system_executor&) noexcept;
```

2 *Returns:* `false`.

13.19 Class `system_context` [async.system.context]

1 Class `system_context` implements the execution context associated with `system_executor` objects.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class system_context : public execution_context
  {
  public:
    // types:

    using executor_type = system_executor;

    // construct / copy / destroy:

    system_context() = delete;
    system_context(const system_context&) = delete;
    system_context& operator=(const system_context&) = delete;
    ~system_context();

    // system_context operations:

    executor_type get_executor() noexcept;

    void stop();
```

```
    bool stopped() const noexcept;
    void join();
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The class `system_context` satisfies the `ExecutionContext` (13.2.3) type requirements.

3 The `system_context` member functions `get_executor`, `stop`, and `stopped`, and the `system_executor` copy constructors, member functions and comparison operators, do not introduce data races as a result of concurrent calls to those functions from different threads of execution.

```
~system_context();
```

4 *Effects:* Performs `stop()` followed by `join()`.

```
executor_type get_executor() noexcept;
```

5 *Returns:* `system_executor()`.

```
void stop();
```

6 *Effects:* Signals all system threads to exit as soon as possible. If a system thread is currently executing a function object, the thread will exit only after completion of that function object. Returns without waiting for the system threads to complete.

7 *Postconditions:* `stopped() == true`.

```
bool stopped() const noexcept;
```

8 *Returns:* `true` if the `system_context` has been stopped by a prior call to `stop`.

```
void join();
```

9 *Effects:* Blocks the calling thread (C++Std [defns.block]) until all system threads have completed.

10 *Synchronization:* The completion of each system thread synchronizes with (C++Std [intro.multithread]) the corresponding successful `join()` return.

13.20 Class `bad_executor` [async.bad.exec]

1 An exception of type `bad_executor` is thrown by `executor` member functions `dispatch`, `post`, and `defer` when the executor object has no target.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class bad_executor : public exception
  {
  public:
    // constructor:
    bad_executor() noexcept;
  };

} // inline namespace v1
```

```
} // namespace net
} // namespace experimental
} // namespace std
```

`bad_executor() noexcept;`

2 *Effects:* constructs a `bad_executor` object.

3 *Postconditions:* `what()` returns an implementation-defined NTBS.

13.21 Class executor [async.executor]

1 The `executor` class provides a polymorphic wrapper for types that satisfy the Executor requirements (13.2.2).

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class executor
  {
  public:
    // construct / copy / destroy:

    executor() noexcept;
    executor(nullptr_t) noexcept;
    executor(const executor& e) noexcept;
    executor(executor&& e) noexcept;
    template<class Executor> executor(Executor e);
    template<class Executor, class ProtoAllocator>
      executor(allocator_arg_t, const ProtoAllocator& a, Executor e);

    executor& operator=(const executor& e) noexcept;
    executor& operator=(executor&& e) noexcept;
    executor& operator=(nullptr_t) noexcept;
    template<class Executor> executor& operator=(Executor e);

    ~executor();

    // executor modifiers:

    void swap(executor& other) noexcept;
    template<class Executor, class ProtoAllocator>
      void assign(Executor e, const ProtoAllocator& a);

    // executor operations:

    execution_context& context() const noexcept;

    void on_work_started() const noexcept;
    void on_work_finished() const noexcept;

    template<class Func, class ProtoAllocator>
      void dispatch(Func&& f, const ProtoAllocator& a) const;
    template<class Func, class ProtoAllocator>
      void post(Func&& f, const ProtoAllocator& a) const;
    template<class Func, class ProtoAllocator>
```

```
      void defer(Func&& f, const ProtoAllocator& a) const;

    // executor capacity:

    explicit operator bool() const noexcept;

    // executor target access:

    const type_info& target_type() const noexcept;
    template<class Executor> Executor* target() noexcept;
    template<class Executor> const Executor* target() const noexcept;
  };

  // executor comparisons:

  bool operator==(const executor& a, const executor& b) noexcept;
  bool operator==(const executor& e, nullptr_t) noexcept;
  bool operator==(nullptr_t, const executor& e) noexcept;
  bool operator!=(const executor& a, const executor& b) noexcept;
  bool operator!=(const executor& e, nullptr_t) noexcept;
  bool operator!=(nullptr_t, const executor& e) noexcept;

  // executor specialized algorithms:

  void swap(executor& a, executor& b) noexcept;

} // inline namespace v1
} // namespace net
} // namespace experimental

  template<class Allocator>
    struct uses_allocator<experimental::net::v1::executor, Allocator>
      : true_type {};

} // namespace std
```

2 Class `executor` meets the requirements of `Executor` (13.2.2), `DefaultConstructible` (C++Std [defaultconstructible]), and `CopyAssignable` (C++Std [copyassignable]).

3 [*Note:* To meet the `noexcept` requirements for executor copy constructors and move constructors, implementations may share a target between two or more `executor` objects. — *end note*]

4 The *target* is the executor object that is held by the wrapper.

13.21.1 executor constructors [async.executor.cons]

`executor() noexcept;`

1 *Postconditions:* `!*this`.

`executor(nullptr_t) noexcept;`

2 *Postconditions:* `!*this`.

`executor(const executor& e) noexcept;`

3 *Postconditions:* `!*this` if `!e`; otherwise, `*this` targets `e.target()` or a copy of `e.target()`.

```
executor(executor&& e) noexcept;
```

4 *Effects:* If `!e`, `*this` has no target; otherwise, moves `e.target()` or move-constructs the target of `e` into the target of `*this`, leaving `e` in a valid state with an unspecified value.

```
template<class Executor> executor(Executor e);
```

5 *Effects:* `*this` targets a copy of `e` initialized with `std::move(e)`.

```
template<class Executor, class ProtoAllocator>
  executor(allocator_arg_t, const ProtoAllocator& a, Executor e);
```

6 *Effects:* `*this` targets a copy of `e` initialized with `std::move(e)`.

7 A copy of the allocator argument is used to allocate memory, if necessary, for the internal data structures of the constructed `executor` object.

13.21.2 executor assignment [async.executor.assign]

```
executor& operator=(const executor& e) noexcept;
```

1 *Effects:* `executor(e).swap(*this)`.

2 *Returns:* `*this`.

```
executor& operator=(executor&& e) noexcept;
```

3 *Effects:* Replaces the target of `*this` with the target of `e`, leaving `e` in a valid state with an unspecified value.

4 *Returns:* `*this`.

```
executor& operator=(nullptr_t) noexcept;
```

5 *Effects:* `executor(nullptr).swap(*this)`.

6 *Returns:* `*this`.

```
template<class Executor> executor& operator=(Executor e);
```

7 *Effects:* `executor(std::move(e)).swap(*this)`.

8 *Returns:* `*this`.

13.21.3 executor destructor [async.executor.dtor]

```
~executor();
```

1 *Effects:* If `*this != nullptr`, releases shared ownership of, or destroys, the target of `*this`.

13.21.4 executor modifiers [async.executor.modifiers]

```
void swap(executor& other) noexcept;
```

1 *Effects:* Interchanges the targets of `*this` and `other`.

```
template<class Executor, class ProtoAllocator>
  void assign(Executor e, const ProtoAllocator& a);
```

2 *Effects:* `executor(allocator_arg, a, std::move(e)).swap(*this)`.

13.21.5 executor operations [async.executor.ops]

```
execution_context& context() const noexcept;
```

1 *Requires:* `*this != nullptr`.

2 *Returns:* `e.context()`, where `e` is the target object of `*this`.

```
void on_work_started() const noexcept;
```

3 *Requires:* `*this != nullptr`.

4 *Effects:* `e.on_work_started()`, where `e` is the target object of `*this`.

```
void on_work_finished() const noexcept;
```

5 *Requires:* `*this != nullptr`.

6 *Effects:* `e.on_work_finished()`, where `e` is the target object of `*this`.

```
template<class Func, class ProtoAllocator>
  void dispatch(Func&& f, const ProtoAllocator& a) const;
```

7 Let `e` be the target object of `*this`. Let `a1` be the allocator that was specified when the target was set. Let `fd` be the result of `DECAY_COPY(f)` (C++Std [thread.decaycopy]).

8 *Effects:* `e.dispatch(g, a1)`, where `g` is a function object of unspecified type that, when called as `g()`, performs `fd()`. The allocator `a` is used to allocate any memory required to implement `g`.

```
template<class Func, class ProtoAllocator>
  void post(Func&& f, const ProtoAllocator& a) const;
```

9 Let `e` be the target object of `*this`. Let `a1` be the allocator that was specified when the target was set. Let `fd` be the result of `DECAY_COPY(f)`.

10 *Effects:* `e.post(g, a1)`, where `g` is a function object of unspecified type that, when called as `g()`, performs `fd()`. The allocator `a` is used to allocate any memory required to implement `g`.

```
template<class Func, class ProtoAllocator>
  void defer(Func&& f, const ProtoAllocator& a) const;
```

11 Let `e` be the target object of `*this`. Let `a1` be the allocator that was specified when the target was set. Let `fd` be the result of `DECAY_COPY(f)`.

12 *Effects:* `e.defer(g, a1)`, where `g` is a function object of unspecified type that, when called as `g()`, performs `fd()`. The allocator `a` is used to allocate any memory required to implement `g`.

13.21.6 executor capacity [async.executor.capacity]

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

1 *Returns:* `true` if `*this` has a target, otherwise `false`.

13.21.7 executor target access [async.executor.target]

```
const type_info& target_type() const noexcept;
```

1 *Returns:* If `*this` has a target of type `T`, `typeid(T)`; otherwise, `typeid(void)`.

```
template<class Executor> Executor* target() noexcept;
template<class Executor> const Executor* target() const noexcept;
```

2 *Returns:* If `target_type() == typeid(Executor)` a pointer to the stored executor target; otherwise a null pointer value.

13.21.8 executor comparisons [async.executor.comparisons]

```
bool operator==(const executor& a, const executor& b) noexcept;
```

1 *Returns:*

(1.1) — `true` if `!a` and `!b`;

(1.2) — `true` if `a` and `b` share a target;

(1.3) — `true` if `e` and `f` are the same type and `e == f`, where `e` is the target of `a` and `f` is the target of `b`;

(1.4) — otherwise `false`.

```
bool operator==(const executor& e, nullptr_t) noexcept;
bool operator==(nullptr_t, const executor& e) noexcept;
```

2 *Returns:* `!e`.

```
bool operator!=(const executor& a, const executor& b) noexcept;
```

3 *Returns:* `!(a == b)`.

```
bool operator!=(const executor& e, nullptr_t) noexcept;
bool operator!=(nullptr_t, const executor& e) noexcept;
```

4 *Returns:* `(bool) e`.

13.21.9 executor specialized algorithms [async.executor.algo]

```
void swap(executor& a, executor& b) noexcept;
```

1 *Effects:* `a.swap(b)`.

13.22 Function dispatch [async.dispatch]

1 [*Note:* The function `dispatch` satisfies the requirements for an asynchronous operation (13.2.7), except for the requirement that the operation uses `post` if it completes immediately. — *end note*]

```
template<class CompletionToken>
  DEDUCED dispatch(CompletionToken&& token);
```

2 *Completion signature:* `void()`.

3 *Effects:*

(3.1) — Constructs an object `completion` of type `async_completion<CompletionToken, void()>`, initialized with `token`.

(3.2) — Performs `ex.dispatch(std::move(completion.completion_handler), alloc)`, where `ex` is the result of `get_associated_executor(completion.completion_handler)`, and `alloc` is the result of `get_associated_allocator(completion.completion_handler)`.

4 *Returns:* `completion.result.get()`.

```
template<class Executor, class CompletionToken>
  DEDUCED dispatch(const Executor& ex, CompletionToken&& token);
```

5 *Completion signature:* `void()`.

6 *Effects:*

(6.1) — Constructs an object `completion` of type `async_completion<CompletionToken, void()>`, initialized with `token`.

(6.2) — Constructs a function object `f` containing as members:

(6.2.1) — a copy of the completion handler `h`, initialized with `std::move(completion.completion_handler)`,

(6.2.2) — an `executor_work_guard` object `w` for the completion handler's associated executor, initialized with `make_work_guard(h)`,

and where the effect of `f()` is:

(6.2.3) — `w.get_executor().dispatch(std::move(h), alloc)`, where `alloc` is the result of `get_associated_allocator(h)`, followed by

(6.2.4) — `w.reset()`.

(6.3) — Performs `ex.dispatch(std::move(f), alloc)`, where `alloc` is the result of `get_associated_allocator(completion.completion_handler)` prior to the construction of `f`.

7 *Returns:* `completion.result.get()`.

8 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<Executor>` is `true`.

```
template<class ExecutionContext, class CompletionToken>
  DEDUCED dispatch(ExecutionContext& ctx, CompletionToken&& token);
```

9 *Completion signature:* `void()`.

10 *Returns:* `std::experimental::net::dispatch(ctx.get_executor(), forward<CompletionToken>(token))`.

11 *Remarks:* This function shall not participate in overload resolution unless `is_convertible<ExecutionContext&, execution_context&>::value` is `true`.

13.23 Function post [async.post]

1 [*Note:* The function `post` satisfies the requirements for an asynchronous operation (13.2.7). — *end note*]

```
template<class CompletionToken>
  DEDUCED post(CompletionToken&& token);
```

2 *Completion signature:* `void()`.

3 *Effects:*

(3.1) — Constructs an object `completion` of type `async_completion<CompletionToken, void()>`, initialized with `token`.

(3.2) — Performs `ex.post(std::move(completion.completion_handler), alloc)`, where `ex` is the result of `get_associated_executor(completion.completion_handler)`, and `alloc` is the result of `get_associated_allocator(completion.completion_handler)`.

4 *Returns:* `completion.result.get()`.

```
template<class Executor, class CompletionToken>
  DEDUCED post(const Executor& ex, CompletionToken&& token);
```

5 *Completion signature:* `void()`.

6 *Effects:*

(6.1) — Constructs an object `completion` of type `async_completion<CompletionToken, void()>`, initialized with `token`.

(6.2) — Constructs a function object `f` containing as members:

(6.2.1) — a copy of the completion handler `h`, initialized with `std::move(completion.completion_handler)`,

(6.2.2) — an `executor_work_guard` object `w` for the completion handler's associated executor, initialized with `make_work_guard(h)`,

and where the effect of `f()` is:

(6.2.3) — `w.get_executor().dispatch(std::move(h), alloc)`, where `alloc` is the result of `get_associated_allocator(h)`, followed by

(6.2.4) — `w.reset()`.

(6.3) — Performs `ex.post(std::move(f), alloc)`, where `alloc` is the result of `get_associated_allocator(completion.completion_handler)` prior to the construction of `f`.

7 *Returns:* `completion.result.get()`.

8 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<Executor>` is `true`.

```
template<class ExecutionContext, class CompletionToken>
  DEDUCED post(ExecutionContext& ctx, CompletionToken&& token);
```

9 *Completion signature:* `void()`.

10 *Returns:* `std::experimental::net::post(ctx.get_executor(), forward<CompletionToken>(token))`.

11 *Remarks:* This function shall not participate in overload resolution unless `is_convertible<ExecutionContext&, execution_context&>::value` is `true`.

13.24 Function `defer` [async.defer]

1 [*Note:* The function `defer` satisfies the requirements for an asynchronous operation (13.2.7), except for the requirement that the operation uses `post` if it completes immediately. — *end note*]

```
template<class CompletionToken>
  DEDUCED defer(CompletionToken&& token);
```

2 *Completion signature:* `void()`.

3 *Effects:*

(3.1) — Constructs an object `completion` of type `async_completion<CompletionToken, void()>`, initialized with `token`.

(3.2) — Performs `ex.defer(std::move(completion.completion_handler), alloc)`, where `ex` is the result of `get_associated_executor(completion.completion_handler)`, and `alloc` is the result of `get_associated_allocator(completion.completion_handler)`.

4 *Returns:* `completion.result.get()`.

```
template<class Executor, class CompletionToken>
  DEDUCED defer(const Executor& ex, CompletionToken&& token);
```

5 *Completion signature:* `void()`.

6 *Effects:*

(6.1) — Constructs an object `completion` of type `async_completion<CompletionToken, void()>`, initialized with `token`.

(6.2) — Constructs a function object `f` containing as members:

(6.2.1) — a copy of the completion handler `h`, initialized with `std::move(completion.completion_handler)`,

(6.2.2) — an `executor_work_guard` object `w` for the completion handler's associated executor, initialized with `make_work_guard(h)`,

and where the effect of `f()` is:

(6.2.3) — `w.get_executor().dispatch(std::move(h), alloc)`, where `alloc` is the result of `get_associated_allocator(h)`, followed by

(6.2.4) — `w.reset()`.

(6.3) — Performs `ex.defer(std::move(f), alloc)`, where `alloc` is the result of `get_associated_allocator(completion.completion_handler)` prior to the construction of `f`.

7 *Returns:* `completion.result.get()`.

8 *Remarks:* This function shall not participate in overload resolution unless `is_executor_v<Executor>` is `true`.

```
template<class ExecutionContext, class CompletionToken>
  DEDUCED defer(ExecutionContext& ctx, CompletionToken&& token);
```

9 *Completion signature:* `void()`.

10 *Returns:* `std::experimental::net::defer(ctx.get_executor(), forward<CompletionToken>(token))`.

11 *Remarks:* This function shall not participate in overload resolution unless `is_convertible<ExecutionContext&, execution_context&>::value` is `true`.

13.25 Class template strand [async.strand]

1 The class template `strand` is a wrapper around an object of type `Executor` satisfying the Executor requirements (13.2.2).

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Executor>
  class strand
  {
  public:
    // types:

    using inner_executor_type = Executor;

    // construct / copy / destroy:

    strand();
    explicit strand(Executor ex);
    template<class ProtoAllocator>
      strand(allocator_arg_t, const ProtoAllocator& alloc, Executor ex);
    strand(const strand& other) noexcept;
    strand(strand&& other) noexcept;
    template<class OtherExecutor> strand(const strand<OtherExecutor>& other) noexcept;
    template<class OtherExecutor> strand(strand<OtherExecutor>&& other) noexcept;

    strand& operator=(const strand& other) noexcept;
```

```
    strand& operator=(strand&& other) noexcept;
    template<class OtherExecutor> strand& operator=(const strand<OtherExecutor>& other) noexcept;
    template<class OtherExecutor> strand& operator=(strand<OtherExecutor>&& other) noexcept;

    ~strand();

    // strand operations:

    inner_executor_type get_inner_executor() const noexcept;

    bool running_in_this_thread() const noexcept;

    execution_context& context() const noexcept;

    void on_work_started() const noexcept;
    void on_work_finished() const noexcept;

    template<class Func, class ProtoAllocator>
      void dispatch(Func&& f, const ProtoAllocator& a) const;
    template<class Func, class ProtoAllocator>
      void post(Func&& f, const ProtoAllocator& a) const;
    template<class Func, class ProtoAllocator>
      void defer(Func&& f, const ProtoAllocator& a) const;

  private:
    Executor inner_ex_; // exposition only
  };

  bool operator==(const strand<Executor>& a, const strand<Executor>& b);
  bool operator!=(const strand<Executor>& a, const strand<Executor>& b);

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 `strand<Executor>` satisfies the `Executor` (13.2.2) requirements.

3 A strand provides guarantees of ordering and non-concurrency. Given:

(3.1) — strand objects `s1` and `s2` such that `s1 == s2`

(3.2) — a function object `f1` added to the strand `s1` using `post` or `defer`, or using `dispatch` when `s1.running_in_this_thread() == false`

(3.3) — a function object `f2` added to the strand `s2` using `post` or `defer`, or using `dispatch` when `s2.running_in_this_thread() == false`

4 then the implementation invokes `f1` and `f2` such that:

(4.1) — the invocation of `f1` is not concurrent with the invocation of `f2`

(4.2) — the invocation of `f1` synchronizes with the invocation of `f2`.

5 Furthermore, if the addition of `f1` happens before the addition of `f2`, then the invocation of `f1` happens before the invocation of `f2`.

6 All member functions, except for the assignment operators and the destructor, do not introduce data races on `*this`, including its ordered, non-concurrent state. Additionally, constructors and assignment operators do not introduce data races on lvalue arguments.

7 If any function `f` executed by the strand throws an exception, the subsequent strand state is as if `f` had exited without throwing an exception.

13.25.1 `strand` constructors [async.strand.cons]

```
strand();
```

1 *Effects:* Constructs an object of class `strand<Executor>` that represents a unique ordered, non-concurrent state. Initializes `inner_ex_` as `inner_ex_()`.

2 *Remarks:* This overload shall not participate in overload resolution unless `Executor` satisfies the `DefaultConstructible` requirements (C++Std [defaultconstructible]).

```
explicit strand(Executor ex);
```

3 *Effects:* Constructs an object of class `strand<Executor>` that represents a unique ordered, non-concurrent state. Initializes `inner_ex_` as `inner_ex_(ex)`.

```
template<class ProtoAllocator>
  strand(allocator_arg_t, const ProtoAllocator& a, Executor ex);
```

4 *Effects:* Constructs an object of class `strand<Executor>` that represents a unique ordered, non-concurrent state. Initializes `inner_ex_` as `inner_ex_(ex)`. A copy of the allocator argument `a` is used to allocate memory, if necessary, for the internal data structures of the constructed strand object.

```
strand(const strand& other) noexcept;
```

5 *Effects:* Initializes `inner_ex_` as `inner_ex_(other.inner_ex_)`.

6 *Postconditions:*

(6.1) — `*this == other`

(6.2) — `get_inner_executor() == other.get_inner_executor()`

```
strand(strand&& other) noexcept;
```

7 *Effects:* Initializes `inner_ex_` as `inner_ex_(std::move(other.inner_ex_))`.

8 *Postconditions:*

(8.1) — `*this` is equal to the prior value of `other`

(8.2) — `get_inner_executor() == other.get_inner_executor()`

```
template<class OtherExecutor> strand(const strand<OtherExecutor>& other) noexcept;
```

9 *Requires:* `OtherExecutor` is convertible to `Executor`.

10 *Effects:* Initializes `inner_ex_` as `inner_ex_(other.inner_ex_)`.

11 *Postconditions:* `*this == other`.

```
template<class OtherExecutor> strand(strand<OtherExecutor>&& other) noexcept;
```

12 *Requires:* `OtherExecutor` is convertible to `Executor`.

13 *Effects:* Initializes `inner_ex_` as `inner_ex_(std::move(other.inner_ex_))`.

14 *Postconditions:* `*this` is equal to the prior value of `other`.

13.25.2 strand assignment [async.strand.assign]

`strand& operator=(const strand& other) noexcept;`

1 *Requires:* `Executor` is `CopyAssignable` (C++Std [copyassignable]).

2 *Postconditions:*

(2.1) — `*this == other`

(2.2) — `get_inner_executor() == other.get_inner_executor()`

3 *Returns:* `*this`.

`strand& operator=(strand&& other) noexcept;`

4 *Requires:* `Executor` is `MoveAssignable` (C++Std [moveassignable]).

5 *Postconditions:*

(5.1) — `*this` is equal to the prior value of `other`

(5.2) — `get_inner_executor() == other.get_inner_executor()`

6 *Returns:* `*this`.

`template<class OtherExecutor> strand& operator=(const strand<OtherExecutor>& other) noexcept;`

7 *Requires:* `OtherExecutor` is convertible to `Executor`. `Executor` is `CopyAssignable` (C++Std [copyassignable]).

8 *Effects:* Assigns `other.inner_ex_` to `inner_ex_`.

9 *Postconditions:* `*this == other`.

10 *Returns:* `*this`.

`template<class OtherExecutor> strand& operator=(strand<OtherExecutor>&& other) noexcept;`

11 *Requires:* `OtherExecutor` is convertible to `Executor`. `Executor` is `MoveAssignable` (C++Std [moveassignable]).

12 *Effects:* Assigns `std::move(other.inner_ex_)` to `inner_ex_`.

13 *Postconditions:* `*this` is equal to the prior value of `other`.

14 *Returns:* `*this`.

13.25.3 strand destructor [async.strand.dtor]

`~strand();`

1 *Effects:* Destroys an object of class `strand<Executor>`. After this destructor completes, objects that were added to the strand but have not yet been executed will be executed in a way that meets the guarantees of ordering and non-concurrency.

13.25.4 strand operations [async.strand.ops]

`inner_executor_type get_inner_executor() const noexcept;`

1 *Returns:* `inner_ex_`.

`bool running_in_this_thread() const noexcept;`

2 *Returns:* `true` if the current thread of execution is running a function that was submitted to the strand, or to any other strand object `s` such that `s == *this`, using `dispatch`, `post` or `defer`; otherwise `false`. [*Note:* That is, the current thread of execution's call chain includes a function that was submitted to the strand. — *end note*]

```
execution_context& context() const noexcept;
```

3 *Returns:* `inner_ex_.context()`.

```
void on_work_started() const noexcept;
```

4 *Effects:* Calls `inner_ex_.on_work_started()`.

```
void on_work_finished() const noexcept;
```

5 *Effects:* Calls `inner_ex_.on_work_finished()`.

```
template<class Func, class ProtoAllocator>
  void dispatch(Func&& f, const ProtoAllocator& a) const;
```

6 *Effects:* If `running_in_this_thread() == true`, calls *`DECAY_COPY`*`(forward<Func>(f))()` (C++Std [thread.decaycopy]). [*Note:* If `f` exits via an exception, the exception propagates to the caller of `dispatch()`. — *end note*] Otherwise, requests invocation of `f`, as if by forwarding the function object `f` and allocator `a` to the executor `inner_ex_`, such that the guarantees of ordering and non-concurrency are met.

```
template<class Func, class ProtoAllocator>
  void post(Func&& f, const ProtoAllocator& a) const;
```

7 *Effects:* Requests invocation of `f`, as if by forwarding the function object `f` and allocator `a` to the executor `inner_ex_`, such that the guarantees of ordering and non-concurrency are met.

```
template<class Func, class ProtoAllocator>
  void defer(Func&& f, const ProtoAllocator& a) const;
```

8 *Effects:* Requests invocation of `f`, as if by forwarding the function object `f` and allocator `a` to the executor `inner_ex_`, such that the guarantees of ordering and non-concurrency are met.

13.25.5 strand comparisons [async.strand.comparisons]

```
bool operator==(const strand<Executor>& a, const strand<Executor>& b);
```

1 *Returns:* `true`, if the strand objects share the same ordered, non-concurrent state; otherwise `false`.

```
bool operator!=(const strand<Executor>& a, const strand<Executor>& b);
```

2 *Returns:* `!(a == b)`.

13.26 Class template use_future_t [async.use.future]

1 The class template `use_future_t` defines a set of types that, when passed as a completion token (13.2.7.2) to an asynchronous operation's initiating function, cause the result of the asynchronous operation to be delivered via a future (C++Std [futures.uniquefuture]).

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class ProtoAllocator = allocator<void>>
```

```
class use_future_t
{
public:
  // use_future_t types:
  using allocator_type = ProtoAllocator;

  // use_future_t members:
  constexpr use_future_t() noexcept(noexcept(allocator_type()));
  explicit use_future_t(const allocator_type& a) noexcept;
  template<class OtherProtoAllocator> use_future_t<OtherProtoAllocator>
    rebind(const OtherProtoAllocator& a) const noexcept;
  allocator_type get_allocator() const noexcept;
  template <class F> unspecified operator()(F&& f) const;
};

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

13.26.1 use_future_t constructors [async.use.future.cons]

`constexpr use_future_t() noexcept(noexcept(allocator_type()));`

1 *Effects:* Constructs a `use_future_t` with a default-constructed allocator.

`explicit use_future_t(const allocator_type& a) noexcept;`

2 *Postconditions:* `get_allocator() == a`.

13.26.2 use_future_t members [async.use.future.members]

```
template<class OtherProtoAllocator> use_future_t<OtherProtoAllocator>
  rebind(const OtherProtoAllocator& a) const noexcept;
```

1 *Returns:* A `use_future_t` object where `get_allocator() == a`.

`allocator_type get_allocator() const noexcept;`

2 *Returns:* The associated allocator object.

`template <class F> unspecified operator()(F&& f) const;`

3 Let `T` be a completion token type. Let `H` be a completion handler type and let `h` be an object of type `H`. Let `FD` be the type `decay_t<F>` and let `fd` be an lvalue of type `FD` constructed with `forward<F>(f)`. Let `R(Args...)` be the completion signature of an asynchronous operation using `H` and let N be `sizeof...(Args)`. Let i be in the range `[0, `N`)` and let $\mathtt{A}_i$ be the i^{th} type in `Args`. Let $\mathtt{a}_i$ be the argument associated with $\mathtt{A}_i$.

4 *Returns:* A completion token `t` of type `T`.

5 *Remarks:* The return type `T` satisfies the `Destructible` (C++Std [destructible]) and `MoveConstructible` (C++Std [moveconstructible]) requirements.

6 The object `h` of type `H` is an asynchronous provider with an associated shared state (C++Std [futures.state]). The effect of `h(`$\mathtt{a}_0$`, ..., `$\mathtt{a}_{N-1}$`)` is to atomically store the result of *`INVOKE`*`(fd, forward<`$\mathtt{A}_0$`>(`$\mathtt{a}_0$`), ..., forward<`$\mathtt{A}_{N-1}$`>(`$\mathtt{a}_{N-1}$`))` (C++Std [func.require]) in the shared state and make the shared state ready. If `fd` exits via an exception then that exception is atomically stored in the shared state and the shared state is made ready.

7 The implementation provides a partial specialization `template <class Result, class... Args> async_result<T, Result(Args...)>` such that:

(7.1) — the nested type `completion_handler_type` is a type `H`;

(7.2) — the nested type `return_type` is `future<result_of_t<FD(decay_t<Args>...)>>`; and

(7.3) — when an object `r1` of type `async_result<T, Result(Args...)>` is constructed from `h`, the expression `r1.get()` returns a future with the same shared state as `h`.

8 For any executor type `E`, the associated object for the associator `associated_executor<H, E>` is an executor where, for function objects executed using the executor's `dispatch()`, `post()` or `defer()` functions, any exception thrown is caught by a function object and stored in the associated shared state.

13.26.3 Partial class template specialization async_result for use_future_t [async.use.future.result]

```
template<class ProtoAllocator, class Result, class... Args>
class async_result<use_future_t<ProtoAllocator>, Result(Args...)>
{
  using completion_handler_type = see below;
  using return_type =  see below;

  explicit async_result(completion_handler_type& h);
  async_result(const async_result&) = delete;
  async_result& operator=(const async_result&) = delete;

  return_type get();
};
```

1 Let `R` be the type `async_result<use_future_t<ProtoAllocator>, Result(Args...)>`. Let `F` be the nested function object type `R::completion_handler_type`.

2 An object `t1` of type `F` is an asynchronous provider with an associated shared state (C++Std [futures.state]). The type `F` provides `F::operator()` such that the expression `t1(declval<Args>()...)` is well formed.

3 The implementation specializes `associated_executor` for `F`. For function objects executed using the associated executor's `dispatch()`, `post()` or `defer()` functions, any exception thrown is caught by the executor and stored in the associated shared state.

4 For any executor type `E`, the associated object for the associator `associated_executor<F, E>` is an executor where, for function objects executed using the executor's `dispatch()`, `post()` or `defer()` functions, any exception thrown by a function object is caught by the executor and stored in the associated shared state.

5 When an object `r1` of type `R` is constructed from `t1`, the expression `r1.get()` returns a future with the same shared state as `t1`.

6 The type of `R::return_type` and the effects of `F::operator()` are defined in Table 10. After establishing these effects, `F::operator()` makes the shared state ready. In this table, N is the value of `sizeof...(Args)`; let i be in the range `[0, N)` and let T_i be the i^{th} type in `Args`; let U_i be `decay_t<T_i>` for each type T_i in `Args`; let A_i be the deduced type of the i^{th} argument to `F::operator()`; and let a_i be the i^{th} argument to `F::operator()`.

Table 10 — async_result<use_future_t<ProtoAllocator>, Result(Args...)> semantics

N	U_0	R::return_type	F::operator() effects
0		future<void>	None.
1	error_code	future<void>	If a_0 evaluates to true, atomically stores the exception pointer produced by make_exception_ptr(system_error(a_0)) in the shared state.
1	exception_ptr	future<void>	If a_0 is non-null, atomically stores the exception pointer a_0 in the shared state.
1	all other types	future<U_0>	Atomically stores forward<A_0>(a_0) in the shared state.
2	error_code	future<U_1>	If a_0 evaluates to true, atomically stores the exception pointer produced by make_exception_ptr(system_error(a_0)) in the shared state; otherwise, atomically stores forward<A_1>(a_1) in the shared state.
2	exception_ptr	future<U_1>	If a_0 is non-null, atomically stores the exception pointer in the shared state; otherwise, atomically stores forward<A_1>(a_1) in the shared state.
2	all other types	future<tuple<U_0, U_1>>	Atomically stores forward_as_tuple(forward<A_0>(a_0), forward<A_1>(a_1)) in the shared state.
>2	error_code	future<tuple<U_1, ..., U_{N-1}>>	If a_0 evaluates to true, atomically stores the exception pointer produced by make_exception_ptr(system_error(a_0)) in the shared state; otherwise, atomically stores forward_as_tuple(forward<A_1>(a_1), ..., forward<A_{N-1}>(a_{N-1})) in the shared state.
>2	exception_ptr	future<tuple<U_1, ..., U_{N-1}>>	If a_0 is non-null, atomically stores the exception pointer in the shared state; otherwise, atomically stores forward_as_tuple(forward<A_1>(a_1), ..., forward<A_{N-1}>(a_{N-1})) in the shared state.
>2	all other types	future<tuple<U_0, ..., U_{N-1}>>	Atomically stores forward_as_tuple(forward<A_0>(a_0), ..., forward<A_{N-1}>(a_{N-1})) in the shared state.

13.27 Partial specialization of async_result for packaged_task [async.packaged.task.spec]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Result, class... Args, class Signature>
  class async_result<packaged_task<Result(Args...)>, Signature>
  {
  public:
    using completion_handler_type = packaged_task<Result(Args...)>;
```

```
    using return_type = future<Result>;

    explicit async_result(completion_handler_type& h);
    async_result(const async_result&) = delete;
    async_result& operator=(const async_result&) = delete;

    return_type get();

  private:
    return_type future_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

`explicit async_result(completion_handler_type& h);`

1 *Effects:* Initializes `future_` with `h.get_future()`.

`return_type get();`

2 *Returns:* `std::move(future_)`.

14 Basic I/O services [io_context]

14.1 Header <experimental/io_context> synopsis [io_context.synop]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class io_context;

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

14.2 Class io_context [io_context.io_context]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class io_context : public execution_context
  {
  public:
    // types:

    class executor_type;
    using count_type = implementation-defined;

    // construct / copy / destroy:

    io_context();
    explicit io_context(int concurrency_hint);
    io_context(const io_context&) = delete;
    io_context& operator=(const io_context&) = delete;

    // io_context operations:

    executor_type get_executor() noexcept;

    count_type run();
    template<class Rep, class Period>
      count_type run_for(const chrono::duration<Rep, Period>& rel_time);
    template<class Clock, class Duration>
      count_type run_until(const chrono::time_point<Clock, Duration>& abs_time);

    count_type run_one();
    template<class Rep, class Period>
      count_type run_one_for(const chrono::duration<Rep, Period>& rel_time);
    template<class Clock, class Duration>
```

```
    count_type run_one_until(const chrono::time_point<Clock, Duration>& abs_time);

  count_type poll();

  count_type poll_one();

  void stop();

  bool stopped() const noexcept;

  void restart();
};

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 The class `io_context` satisfies the ExecutionContext type requirements (13.2.3).

2 `count_type` is an implementation-defined unsigned integral type of at least 32 bits.

3 The `io_context` member functions `run`, `run_for`, `run_until`, `run_one`, `run_one_for`, `run_one_until`, `poll`, and `poll_one` are collectively referred to as the *run functions*. The run functions must be called for the `io_context` to perform asynchronous operations (3.4) on behalf of a C++ program. Notification that an asynchronous operation has completed is delivered by execution of the associated completion handler function object, as determined by the requirements for asynchronous operations (13.2.7).

4 For an object of type `io_context`, *outstanding work* is defined as the sum of:

(4.1) — the total number of calls to the `on_work_started` function, less the total number of calls to the `on_work_finished` function, to any executor of the `io_context`.

(4.2) — the number of function objects that have been added to the `io_context` via any executor of the `io_context`, but not yet executed; and

(4.3) — the number of function objects that are currently being executed by the `io_context`.

5 If at any time the outstanding work falls to `0`, the `io_context` is stopped as if by `stop()`.

6 The `io_context` member functions `get_executor`, `stop`, and `stopped`, the run functions, and the `io_context::executor_type` copy constructors, member functions and comparison operators, do not introduce data races as a result of concurrent calls to those functions from different threads of execution. [*Note:* The `restart` member function is excluded from these thread safety requirements. — *end note*]

14.2.1 `io_context` members [io_context.io_context.members]

```
io_context();
explicit io_context(int concurrency_hint);
```

1 *Effects:* Creates an object of class `io_context`.

2 *Remarks:* The `concurrency_hint` parameter is a suggestion to the implementation on the number of threads that should process asynchronous operations and execute function objects.

```
executor_type get_executor() noexcept;
```

3 *Returns:* An executor that may be used for submitting function objects to the `io_context`.

```
count_type run();
```

4 *Requires:* Must not be called from a thread that is currently calling a run function.

5 *Effects:* Equivalent to:

```
count_type n = 0;
while (run_one())
  if (n != numeric_limits<count_type>::max())
    ++n;
```

6 *Returns:* `n`.

```
template<class Rep, class Period>
  count_type run_for(const chrono::duration<Rep, Period>& rel_time);
```

7 *Effects:* Equivalent to:

```
return run_until(chrono::steady_clock::now() + rel_time);
```

```
template<class Clock, class Duration>
  count_type run_until(const chrono::time_point<Clock, Duration>& abs_time);
```

8 *Effects:* Equivalent to:

```
count_type n = 0;
while (run_one_until(abs_time))
  if (n != numeric_limits<count_type>::max())
    ++n;
```

9 *Returns:* `n`.

```
count_type run_one();
```

10 *Requires:* Must not be called from a thread that is currently calling a run function.

11 *Effects:* If the `io_context` object has no outstanding work, performs `stop()`. Otherwise, blocks while the io_context has outstanding work, or until the `io_context` is stopped, or until one function object has been executed.

12 If an executed function object throws an exception, the exception propagates to the caller of `run_one()`. The `io_context` state is as if the function object had returned normally.

13 *Returns:* `1` if a function object was executed, otherwise `0`.

14 Notes: This function may invoke additional function objects through nested calls to the `io_context` executor's `dispatch` member function. These do not count towards the return value.

```
template<class Rep, class Period>
  count_type run_one_for(const chrono::duration<Rep, Period>& rel_time);
```

15 *Effects:* Equivalent to:

```
return run_one_until(chrono::steady_clock::now() + rel_time);
```

```
template<class Clock, class Duration>
  count_type run_one_until(const chrono::time_point<Clock, Duration>& abs_time);
```

16 *Effects:* If the `io_context` object has no outstanding work, performs `stop()`. Otherwise, blocks while the io_context has outstanding work, or until the expiration of the absolute timeout (C++Std [thread.req.timing]) specified by `abs_time`, or until the `io_context` is stopped, or until one function object has been executed.

17 If an executed function object throws an exception, the exception propagates to the caller of `run_one()`. The `io_context` state is as if the function object had returned normally.

18 *Returns:* `1` if a function object was executed, otherwise `0`.

19 Notes: This function may invoke additional function objects through nested calls to the `io_context` executor's `dispatch` member function. These do not count towards the return value.

```
count_type poll();
```

20 *Effects:* Equivalent to:

```
count_type n = 0;
while (poll_one())
  if (n != numeric_limits<count_type>::max())
    ++n;
```

21 *Returns:* `n`.

```
count_type poll_one();
```

22 *Effects:* If the `io_context` object has no outstanding work, performs `stop()`. Otherwise, if there is a function object ready for immediate execution, executes it.

23 If an executed function object throws an exception, the exception propagates to the caller of `poll_one()`. The `io_context` state is as if the function object had returned normally.

24 *Returns:* `1` if a function object was invoked, otherwise `0`.

25 Notes: This function may invoke additional function objects through nested calls to the `io_context` executor's `dispatch` member function. These do not count towards the return value.

```
void stop();
```

26 *Effects:* Stops the `io_context`. Concurrent calls to any run function will end as soon as possible. If a call to a run function is currently executing a function object, the call will end only after completion of that function object. The call to `stop()` returns without waiting for concurrent calls to run functions to complete.

27 *Postconditions:* `stopped() == true`.

28 [*Note:* When `stopped() == true`, subsequent calls to a run function will exit immediately with a return value of `0`, without executing any function objects. An `io_context` remains in the stopped state until a call to `restart()`. — *end note*]

```
bool stopped() const noexcept;
```

29 *Returns:* `true` if the `io_context` is stopped.

```
void restart();
```

30 *Postconditions:* `stopped() == false`.

14.3 Class `io_context::executor_type` [io_context.exec]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class io_context::executor_type
  {
```

```
  public:
    // construct / copy / destroy:

    executor_type(const executor_type& other) noexcept;
    executor_type(executor_type&& other) noexcept;

    executor_type& operator=(const executor_type& other) noexcept;
    executor_type& operator=(executor_type&& other) noexcept;

    // executor operations:

    bool running_in_this_thread() const noexcept;

    io_context& context() const noexcept;

    void on_work_started() const noexcept;
    void on_work_finished() const noexcept;

    template<class Func, class ProtoAllocator>
      void dispatch(Func&& f, const ProtoAllocator& a) const;
    template<class Func, class ProtoAllocator>
      void post(Func&& f, const ProtoAllocator& a) const;
    template<class Func, class ProtoAllocator>
      void defer(Func&& f, const ProtoAllocator& a) const;
  };

  bool operator==(const io_context::executor_type& a,
                  const io_context::executor_type& b) noexcept;
  bool operator!=(const io_context::executor_type& a,
                  const io_context::executor_type& b) noexcept;

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 `io_context::executor_type` is a type satisfying the Executor requirements (13.2.2). Objects of type `io_context::executor_type` are associated with an `io_context`, and function objects submitted using the `dispatch`, `post`, or `defer` member functions will be executed by the `io_context` from within a run function.]

14.3.1 io_context::executor_type constructors [io_context.exec.cons]

`executor_type(const executor_type& other) noexcept;`

1 *Postconditions:* `*this == other`.

`executor_type(executor_type&& other) noexcept;`

2 *Postconditions:* `*this` is equal to the prior value of `other`.

14.3.2 io_context::executor_type assignment [io_context.exec.assign]

`executor_type& operator=(const executor_type& other) noexcept;`

1 *Postconditions:* `*this == other`.

2 *Returns:* `*this`.

`executor_type& operator=(executor_type&& other) noexcept;`

3 *Postconditions:* `*this` is equal to the prior value of `other`.

4 *Returns:* `*this`.

14.3.3 `io_context::executor_type` operations [io_context.exec.ops]

```
bool running_in_this_thread() const noexcept;
```

1 *Returns:* `true` if the current thread of execution is calling a run function of the associated `io_context` object. [*Note:* That is, the current thread of execution's call chain includes a run function. — *end note*]

```
io_context& context() const noexcept;
```

2 *Returns:* A reference to the associated `io_context` object.

```
void on_work_started() const noexcept;
```

3 *Effects:* Increments the count of outstanding work associated with the `io_context`.

```
void on_work_finished() const noexcept;
```

4 *Effects:* Decrements the count of outstanding work associated with the `io_context`.

```
template<class Func, class ProtoAllocator>
  void dispatch(Func&& f, const ProtoAllocator& a) const;
```

5 *Effects:* If `running_in_this_thread()` is `true`, calls *`DECAY_COPY`*`(forward<Func>(f))()` (C++Std [thread.decaycopy]). [*Note:* If `f` exits via an exception, the exception propagates to the caller of `dispatch()`. — *end note*] Otherwise, calls `post(forward<Func>(f), a)`.

```
template<class Func, class ProtoAllocator>
  void post(Func&& f, const ProtoAllocator& a) const;
```

6 *Effects:* Adds `f` to the `io_context`.

```
template<class Func, class ProtoAllocator>
  void defer(Func&& f, const ProtoAllocator& a) const;
```

7 *Effects:* Adds `f` to the `io_context`.

14.3.4 `io_context::executor_type` comparisons [io_context.exec.comparisons]

```
bool operator==(const io_context::executor_type& a,
                const io_context::executor_type& b) noexcept;
```

1 *Returns:* `addressof(a.context()) == addressof(b.context())`.

```
bool operator!=(const io_context::executor_type& a,
                const io_context::executor_type& b) noexcept;
```

2 *Returns:* `!(a == b)`.

15 Timers [timer]

1 This clause defines components for performing timer operations.

2 [*Example:* Performing a synchronous wait operation on a timer:

```
io_context c;
steady_timer t(c);
t.expires_after(seconds(5));
t.wait();
```

— *end example*]

3 [*Example:* Performing an asynchronous wait operation on a timer:

```
void handler(error_code ec) { ... }
...
io_context c;
steady_timer t(c);
t.expires_after(seconds(5));
t.async_wait(handler);
c.run();
```

— *end example*]

15.1 Header `<experimental/timer>` synopsis [timer.synop]

```
#include <chrono>

namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Clock> struct wait_traits;

  template<class Clock, class WaitTraits = wait_traits<Clock>>
    class basic_waitable_timer;

  using system_timer = basic_waitable_timer<chrono::system_clock>;
  using steady_timer = basic_waitable_timer<chrono::steady_clock>;
  using high_resolution_timer = basic_waitable_timer<chrono::high_resolution_clock>;

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

15.2 Requirements [timer.reqmts]

15.2.1 Wait traits requirements [timer.reqmts.waittraits]

1 The `basic_waitable_timer` template uses wait traits to allow programs to customize `wait` and `async_wait` behavior. [*Note:* Possible uses of wait traits include:

(1.1) — To enable timers based on non-realtime clocks.

(1.2) — Determining how quickly wallclock-based timers respond to system time changes.

(1.3) — Correcting for errors or rounding timeouts to boundaries.

(1.4) — Preventing duration overflow. That is, a program may set a timer's expiry `e` to be `Clock::max()` (meaning never reached) or `Clock::min()` (meaning always in the past). As a result, computing the duration until timer expiry as `e - Clock::now()` may cause overflow.

— *end note*]

2 For a type `Clock` meeting the `Clock` requirements (C++Std [time.clock.req]), a type `X` meets the `WaitTraits` requirements if it satisfies the requirements listed below.

3 In Table 11, `t` denotes a (possibly const) value of type `Clock::time_point`; and `d` denotes a (possibly const) value of type `Clock::duration`.

Table 11 — WaitTraits requirements

expression	return type	assertion/note pre/post-condition
`X::to_wait_duration(d)`	`Clock::duration`	Returns a `Clock::duration` value to be used in a `wait` or `async_wait` operation. [*Note:* The return value is typically representative of the duration `d`. — *end note*]
`X::to_wait_duration(t)`	`Clock::duration`	Returns a `Clock::duration` value to be used in a `wait` or `async_wait` operation. [*Note:* The return value is typically representative of the duration from `Clock::now()` until the time point `t`. — *end note*]

15.3 Class template `wait_traits` [timer.waittraits]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Clock>
  struct wait_traits
  {
    static typename Clock::duration to_wait_duration(
      const typename Clock::duration& d);

    static typename Clock::duration to_wait_duration(
      const typename Clock::time_point& t);
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 Class template `wait_traits` satisfies the `WaitTraits` (15.2.1) type requirements. Template argument `Clock` is a type meeting the `Clock` requirements (C++Std [time.clock.req]).

```
static typename Clock::duration to_wait_duration(
  const typename Clock::duration& d);
```

2 *Returns:* `d`.

```
static typename Clock::duration to_wait_duration(
  const typename Clock::time_point& t);
```

3 *Returns:* Let `now` be `Clock::now()`. If `now + Clock::duration::max()` is before `t`, `Clock::duration::max()`; if `now + Clock::duration::min()` is after `t`, `Clock::duration::min()`; otherwise, `t - now`.

15.4 Class template `basic_waitable_timer` [timer.waitable]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Clock, class WaitTraits = wait_traits<Clock>>
  class basic_waitable_timer
  {
  public:
    // types:

    using executor_type = io_context::executor_type;
    using clock_type = Clock;
    using duration = typename clock_type::duration;
    using time_point = typename clock_type::time_point;
    using traits_type = WaitTraits;

    // construct / copy / destroy:

    explicit basic_waitable_timer(io_context& ctx);
    basic_waitable_timer(io_context& ctx, const time_point& t);
    basic_waitable_timer(io_context& ctx, const duration& d);
    basic_waitable_timer(const basic_waitable_timer&) = delete;
    basic_waitable_timer(basic_waitable_timer&& rhs);

    ~basic_waitable_timer();

    basic_waitable_timer& operator=(const basic_waitable_timer&) = delete;
    basic_waitable_timer& operator=(basic_waitable_timer&& rhs);

    // basic_waitable_timer operations:

    executor_type get_executor() noexcept;

    size_t cancel();
    size_t cancel_one();

    time_point expiry() const;
    size_t expires_at(const time_point& t);
    size_t expires_after(const duration& d);

    void wait();
    void wait(error_code& ec);
```

```
    template<class CompletionToken>
      DEDUCED async_wait(CompletionToken&& token);
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 Instances of class template `basic_waitable_timer` meet the requirements of `Destructible` (C++Std [destructible]), `MoveConstructible` (C++Std [moveconstructible]), and `MoveAssignable` (C++Std [moveassignable]).

15.4.1 `basic_waitable_timer` constructors [timer.waitable.cons]

`explicit basic_waitable_timer(io_context& ctx);`

1 *Effects:* Equivalent to `basic_waitable_timer(ctx, time_point())`.

`basic_waitable_timer(io_context& ctx, const time_point& t);`

2 *Postconditions:*

(2.1) — `get_executor() == ctx.get_executor()`.

(2.2) — `expiry() == t`.

`basic_waitable_timer(io_context& ctx, const duration& d);`

3 *Effects:* Sets the expiry time as if by calling `expires_after(d)`.

4 *Postconditions:* `get_executor() == ctx.get_executor()`.

`basic_waitable_timer(basic_waitable_timer&& rhs);`

5 *Effects:* Move constructs an object of class `basic_waitable_timer<Clock, WaitTraits>` that refers to the state originally represented by `rhs`.

6 *Postconditions:*

(6.1) — `get_executor() == rhs.get_executor()`.

(6.2) — `expiry()` returns the same value as `rhs.expiry()` prior to the constructor invocation.

(6.3) — `rhs.expiry() == time_point()`.

15.4.2 `basic_waitable_timer` destructor [timer.waitable.dtor]

`~basic_waitable_timer();`

1 *Effects:* Destroys the timer, canceling any asynchronous wait operations associated with the timer as if by calling `cancel()`.

15.4.3 `basic_waitable_timer` assignment [timer.waitable.assign]

`basic_waitable_timer& operator=(basic_waitable_timer&& rhs);`

1 *Effects:* Cancels any outstanding asynchronous operations associated with `*this` as if by calling `cancel()`, then moves into `*this` the state originally represented by `rhs`.

2 *Postconditions:*

(2.1) — `get_executor() == rhs.get_executor()`.

(2.2) — `expiry()` returns the same value as `rhs.expiry()` prior to the assignment.

(2.3) — `rhs.expiry() == time_point()`.

3 *Returns:* `*this`.

15.4.4 `basic_waitable_timer` operations [timer.waitable.ops]

```
executor_type get_executor() noexcept;
```

1 *Returns:* The associated executor.

```
size_t cancel();
```

2 *Effects:* Causes any outstanding asynchronous wait operations to complete. Completion handlers for canceled operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

3 *Returns:* The number of operations that were canceled.

4 *Remarks:* Does not block (C++Std [defns.block]) the calling thread pending completion of the canceled operations.

```
size_t cancel_one();
```

5 *Effects:* Causes the outstanding asynchronous wait operation that was initiated first, if any, to complete as soon as possible. The completion handler for the canceled operation is passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

6 *Returns:* `1` if an operation was canceled, otherwise `0`.

7 *Remarks:* Does not block (C++Std [defns.block]) the calling thread pending completion of the canceled operation.

```
time_point expiry() const;
```

8 *Returns:* The expiry time associated with the timer, as previously set using `expires_at()` or `expires_after()`.

```
size_t expires_at(const time_point& t);
```

9 *Effects:* Cancels outstanding asynchronous wait operations, as if by calling `cancel()`. Sets the expiry time associated with the timer.

10 *Returns:* The number of operations that were canceled.

11 *Postconditions:* `expiry() == t`.

```
size_t expires_after(const duration& d);
```

12 *Returns:* `expires_at(clock_type::now() + d)`.

```
void wait();
void wait(error_code& ec);
```

13 *Effects:* Establishes the postcondition as if by repeatedly blocking the calling thread (C++Std [defns.block]) for the relative time produced by `WaitTraits::to_wait_duration(expiry())`.

14 *Postconditions:* `ec || expiry() <= clock_type::now()`.

```
template<class CompletionToken>
  DEDUCED async_wait(CompletionToken&& token);
```

15 *Completion signature:* void(error_code ec).

16 *Effects:* Initiates an asynchronous wait operation to repeatedly wait for the relative time produced by WaitTraits::to_wait_duration(e), where e is a value of type time_point such that e <= expiry(). The completion handler is submitted for execution only when the condition ec || expiry() <= clock_type::now() yields true.

17 [*Note:* To implement async_wait, an io_context object ctx may maintain a priority queue for each specialization of basic_waitable_timer<Clock, WaitTraits> for which a timer object was initialized with ctx. Only the time point e of the earliest outstanding expiry need be passed to WaitTraits::to_wait_duration(e). — *end note*]

16 Buffers [buffer]

16.1 Header <experimental/buffer> synopsis [buffer.synop]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  enum class stream_errc {
    eof = an implementation defined non-zero value,
    not_found = an implementation defined non-zero value
  };

  const error_category& stream_category() noexcept;

  error_code make_error_code(stream_errc e) noexcept;
  error_condition make_error_condition(stream_errc e) noexcept;

  class mutable_buffer;
  class const_buffer;

  // buffer type traits:

  template<class T> struct is_mutable_buffer_sequence;
  template<class T> struct is_const_buffer_sequence;
  template<class T> struct is_dynamic_buffer;

  template<class T>
      constexpr bool is_mutable_buffer_sequence_v = is_mutable_buffer_sequence<T>::value;
    template<class T>
      constexpr bool is_const_buffer_sequence_v = is_const_buffer_sequence<T>::value;
    template<class T>
      constexpr bool is_dynamic_buffer_v = is_dynamic_buffer<T>::value;

  // buffer sequence access:

  const mutable_buffer* buffer_sequence_begin(const mutable_buffer& b);
  const const_buffer* buffer_sequence_begin(const const_buffer& b);
  const mutable_buffer* buffer_sequence_end(const mutable_buffer& b);
  const const_buffer* buffer_sequence_end(const const_buffer& b);
  template<class C> auto buffer_sequence_begin(C& c) -> decltype(c.begin());
  template<class C> auto buffer_sequence_begin(const C& c) -> decltype(c.begin());
  template<class C> auto buffer_sequence_end(C& c) -> decltype(c.end());
  template<class C> auto buffer_sequence_end(const C& c) -> decltype(c.end());

  // buffer size:

  template<class ConstBufferSequence>
    size_t buffer_size(const ConstBufferSequence& buffers) noexcept;

  // buffer copy:
```

```
template<class MutableBufferSequence, class ConstBufferSequence>
  size_t buffer_copy(const MutableBufferSequence& dest,
                     const ConstBufferSequence& source) noexcept;
template<class MutableBufferSequence, class ConstBufferSequence>
  size_t buffer_copy(const MutableBufferSequence& dest,
                     const ConstBufferSequence& source,
                     size_t max_size) noexcept;

// buffer arithmetic:

mutable_buffer operator+(const mutable_buffer& b, size_t n) noexcept;
mutable_buffer operator+(size_t n, const mutable_buffer& b) noexcept;
const_buffer operator+(const const_buffer&, size_t n) noexcept;
const_buffer operator+(size_t, const const_buffer&) noexcept;

// buffer creation:

mutable_buffer buffer(void* p, size_t n) noexcept;
const_buffer buffer(const void* p, size_t n) noexcept;

mutable_buffer buffer(const mutable_buffer& b) noexcept;
mutable_buffer buffer(const mutable_buffer& b, size_t n) noexcept;
const_buffer buffer(const const_buffer& b) noexcept;
const_buffer buffer(const const_buffer& b, size_t n) noexcept;

template<class T, size_t N>
  mutable_buffer buffer(T (&data)[N]) noexcept;
template<class T, size_t N>
  const_buffer buffer(const T (&data)[N]) noexcept;
template<class T, size_t N>
  mutable_buffer buffer(array<T, N>& data) noexcept;
template<class T, size_t N>
  const_buffer buffer(array<const T, N>& data) noexcept;
template<class T, size_t N>
  const_buffer buffer(const array<T, N>& data) noexcept;
template<class T, class Allocator>
  mutable_buffer buffer(vector<T, Allocator>& data) noexcept;
template<class T, class Allocator>
  const_buffer buffer(const vector<T, Allocator>& data) noexcept;
template<class CharT, class Traits, class Allocator>
  mutable_buffer buffer(basic_string<CharT, Traits, Allocator>& data) noexcept;
template<class CharT, class Traits, class Allocator>
  const_buffer buffer(const basic_string<CharT, Traits, Allocator>& data) noexcept;
template<class CharT, class Traits>
  const_buffer buffer(basic_string_view<CharT, Traits> data) noexcept;

template<class T, size_t N>
  mutable_buffer buffer(T (&data)[N], size_t n) noexcept;
template<class T, size_t N>
  const_buffer buffer(const T (&data)[N], size_t n) noexcept;
template<class T, size_t N>
  mutable_buffer buffer(array<T, N>& data, size_t n) noexcept;
template<class T, size_t N>
  const_buffer buffer(array<const T, N>& data, size_t n) noexcept;
```

```
template<class T, size_t N>
  const_buffer buffer(const array<T, N>& data, size_t n) noexcept;
template<class T, class Allocator>
  mutable_buffer buffer(vector<T, Allocator>& data, size_t n) noexcept;
template<class T, class Allocator>
  const_buffer buffer(const vector<T, Allocator>& data, size_t n) noexcept;
template<class CharT, class Traits, class Allocator>
  mutable_buffer buffer(basic_string<CharT, Traits, Allocator>& data,
                        size_t n) noexcept;
template<class CharT, class Traits, class Allocator>
  const_buffer buffer(const basic_string<CharT, Traits, Allocator>& data,
                      size_t n) noexcept;
template<class CharT, class Traits>
  const_buffer buffer(basic_string_view<CharT, Traits> data,
                      size_t n) noexcept;

template<class T, class Allocator>
  class dynamic_vector_buffer;

template<class CharT, class Traits, class Allocator>
  class dynamic_string_buffer;

// dynamic buffer creation:

template<class T, class Allocator>
  dynamic_vector_buffer<T, Allocator>
  dynamic_buffer(vector<T, Allocator>& vec) noexcept;
template<class T, class Allocator>
  dynamic_vector_buffer<T, Allocator>
  dynamic_buffer(vector<T, Allocator>& vec, size_t n) noexcept;

template<class CharT, class Traits, class Allocator>
  dynamic_string_buffer<CharT, Traits, Allocator>
  dynamic_buffer(basic_string<CharT, Traits, Allocator>& str) noexcept;
template<class CharT, class Traits, class Allocator>
  dynamic_string_buffer<CharT, Traits, Allocator>
  dynamic_buffer(basic_string<CharT, Traits, Allocator>& str, size_t n) noexcept;

class transfer_all;
class transfer_at_least;
class transfer_exactly;

// synchronous read operations:

template<class SyncReadStream, class MutableBufferSequence>
  size_t read(SyncReadStream& stream,
              const MutableBufferSequence& buffers);
template<class SyncReadStream, class MutableBufferSequence>
  size_t read(SyncReadStream& stream,
              const MutableBufferSequence& buffers, error_code& ec);
template<class SyncReadStream, class MutableBufferSequence,
  class CompletionCondition>
    size_t read(SyncReadStream& stream,
                const MutableBufferSequence& buffers,
                CompletionCondition completion_condition);
```

```
template<class SyncReadStream, class MutableBufferSequence,
  class CompletionCondition>
    size_t read(SyncReadStream& stream,
                const MutableBufferSequence& buffers,
                CompletionCondition completion_condition,
                error_code& ec);

template<class SyncReadStream, class DynamicBuffer>
  size_t read(SyncReadStream& stream, DynamicBuffer&& b);
template<class SyncReadStream, class DynamicBuffer>
  size_t read(SyncReadStream& stream, DynamicBuffer&& b, error_code& ec);
template<class SyncReadStream, class DynamicBuffer, class CompletionCondition>
  size_t read(SyncReadStream& stream, DynamicBuffer&& b,
              CompletionCondition completion_condition);
template<class SyncReadStream, class DynamicBuffer, class CompletionCondition>
  size_t read(SyncReadStream& stream, DynamicBuffer&& b,
              CompletionCondition completion_condition, error_code& ec);

// asynchronous read operations:

template<class AsyncReadStream, class MutableBufferSequence,
  class CompletionToken>
    DEDUCED async_read(AsyncReadStream& stream,
                       const MutableBufferSequence& buffers,
                       CompletionToken&& token);
template<class AsyncReadStream, class MutableBufferSequence,
  class CompletionCondition, class CompletionToken>
    DEDUCED async_read(AsyncReadStream& stream,
                       const MutableBufferSequence& buffers,
                       CompletionCondition completion_condition,
                       CompletionToken&& token);

template<class AsyncReadStream, class DynamicBuffer, class CompletionToken>
  DEDUCED async_read(AsyncReadStream& stream,
                     DynamicBuffer&& b, CompletionToken&& token);
template<class AsyncReadStream, class DynamicBuffer,
  class CompletionCondition, class CompletionToken>
    DEDUCED async_read(AsyncReadStream& stream,
                       DynamicBuffer&& b,
                       CompletionCondition completion_condition,
                       CompletionToken&& token);

// synchronous write operations:

template<class SyncWriteStream, class ConstBufferSequence>
  size_t write(SyncWriteStream& stream,
               const ConstBufferSequence& buffers);
template<class SyncWriteStream, class ConstBufferSequence>
  size_t write(SyncWriteStream& stream,
               const ConstBufferSequence& buffers, error_code& ec);
template<class SyncWriteStream, class ConstBufferSequence,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 const ConstBufferSequence& buffers,
                 CompletionCondition completion_condition);
```

```
template<class SyncWriteStream, class ConstBufferSequence,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 const ConstBufferSequence& buffers,
                 CompletionCondition completion_condition,
                 error_code& ec);

template<class SyncWriteStream, class DynamicBuffer>
  size_t write(SyncWriteStream& stream, DynamicBuffer&& b);
template<class SyncWriteStream, class DynamicBuffer>
  size_t write(SyncWriteStream& stream, DynamicBuffer&& b, error_code& ec);
template<class SyncWriteStream, class DynamicBuffer, class CompletionCondition>
  size_t write(SyncWriteStream& stream, DynamicBuffer&& b,
               CompletionCondition completion_condition);
template<class SyncWriteStream, class DynamicBuffer, class CompletionCondition>
  size_t write(SyncWriteStream& stream, DynamicBuffer&& b,
               CompletionCondition completion_condition, error_code& ec);

// asynchronous write operations:

template<class AsyncWriteStream, class ConstBufferSequence,
  class CompletionToken>
    DEDUCED async_write(AsyncWriteStream& stream,
                        const ConstBufferSequence& buffers,
                        CompletionToken&& token);
template<class AsyncWriteStream, class ConstBufferSequence,
  class CompletionCondition, class CompletionToken>
    DEDUCED async_write(AsyncWriteStream& stream,
                        const ConstBufferSequence& buffers,
                        CompletionCondition completion_condition,
                        CompletionToken&& token);

template<class AsyncWriteStream, class DynamicBuffer, class CompletionToken>
  DEDUCED async_write(AsyncWriteStream& stream,
                   DynamicBuffer&& b, CompletionToken&& token);
template<class AsyncWriteStream, class DynamicBuffer,
  class CompletionCondition, class CompletionToken>
    DEDUCED async_write(AsyncWriteStream& stream,
                        DynamicBuffer&& b,
                        CompletionCondition completion_condition,
                        CompletionToken&& token);

// synchronous delimited read operations:

template<class SyncReadStream, class DynamicBuffer>
  size_t read_until(SyncReadStream& s, DynamicBuffer&& b, char delim);
template<class SyncReadStream, class DynamicBuffer>
  size_t read_until(SyncReadStream& s, DynamicBuffer&& b,
                    char delim, error_code& ec);
template<class SyncReadStream, class DynamicBuffer>
  size_t read_until(SyncReadStream& s, DynamicBuffer&& b, string_view delim);
template<class SyncReadStream, class DynamicBuffer>
  size_t read_until(SyncReadStream& s, DynamicBuffer&& b,
                    string_view delim, error_code& ec);
```

```
  // asynchronous delimited read operations:

  template<class AsyncReadStream, class DynamicBuffer, class CompletionToken>
    DEDUCED async_read_until(AsyncReadStream& s,
                             DynamicBuffer&& b, char delim,
                             CompletionToken&& token);
  template<class AsyncReadStream, class DynamicBuffer, class CompletionToken>
    DEDUCED async_read_until(AsyncReadStream& s,
                             DynamicBuffer&& b, string_view delim,
                             CompletionToken&& token);

} // inline namespace v1
} // namespace net
} // namespace experimental

  template<> struct is_error_code_enum<
    experimental::net::v1::stream_errc>
      : public true_type {};

} // namespace std
```

16.2 Requirements [buffer.reqmts]

16.2.1 Mutable buffer sequence requirements [buffer.reqmts.mutablebuffersequence]

1 A *mutable buffer sequence* represents a set of memory regions that may be used to receive the output of an operation, such as the `receive` operation of a socket.

2 A type `X` meets the `MutableBufferSequence` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `CopyConstructible` (C++Std [copyconstructible]), as well as the additional requirements listed in Table 12.

3 In Table 12, `x` denotes a (possibly const) value of type `X`, and `u` denotes an identifier.

Table 12 — MutableBufferSequence requirements

expression	return type	assertion/note pre/post-condition
`net::buffer_sequence_begin(x)` `net::buffer_sequence_end(x)`	An iterator type meeting the requirements for bidirectional iterators (C++Std [bidirectional.iterators]) whose value type is convertible to `mutable_buffer`.	

Table 12 — MutableBufferSequence requirements (continued)

expression	return type	assertion/note pre/post-condition

`X u(x);`		post:
		`equal(`
		`  net::buffer_sequence_begin(x),`
		`  net::buffer_sequence_end(x),`
		`  net::buffer_sequence_begin(u),`
		`  net::buffer_sequence_end(u),`
		`  [](const typename X::value_type& v1,`
		`     const typename X::value_type& v2)`
		`   {`
		`     mutable_buffer b1(v1);`
		`     mutable_buffer b2(v2);`
		`     return b1.data() == b2.data()`
		`         && b1.size() == b2.size();`
		`   })`

16.2.2 Constant buffer sequence requirements [buffer.reqmts.constbuffersequence]

1 A *constant buffer sequence* represents a set of memory regions that may be used as input to an operation, such as the `send` operation of a socket.

2 A type `X` meets the `ConstBufferSequence` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `CopyConstructible` (C++Std [copyconstructible]), as well as the additional requirements listed in Table 13.

3 In Table 13, `x` denotes a (possibly const) value of type `X`, and `u` denotes an identifier.

Table 13 — ConstBufferSequence requirements

expression	return type	assertion/note pre/post-condition
`net::buffer_sequence_begin(x)` `net::buffer_sequence_end(x)`	An iterator type meeting the requirements for bidirectional iterators (C++Std [bidirectional.iterators]) whose value type is convertible to `const_buffer`.	

Table 13 — ConstBufferSequence requirements (continued)

expression	return type	assertion/note pre/post-condition

`X u(x);`		post:
		`equal(`
		`  net::buffer_sequence_begin(x),`
		`  net::buffer_sequence_end(x),`
		`  net::buffer_sequence_begin(u),`
		`  net::buffer_sequence_end(u),`
		`  [](const typename X::value_type& v1,`
		`     const typename X::value_type& v2)`
		`   {`
		`     const_buffer b1(v1);`
		`     const_buffer b2(v2);`
		`     return b1.data() == b2.data()`
		`         && b1.size() == b2.size();`
		`   })`

16.2.3 Dynamic buffer requirements [buffer.reqmts.dynamicbuffer]

1 A *dynamic buffer* encapsulates memory storage that may be automatically resized as required, where the memory is divided into two regions: readable bytes followed by writable bytes. These memory regions are internal to the dynamic buffer, but direct access to the elements is provided to permit them to be efficiently used with I/O operations. [*Note:* Such as the `send` or `receive` operations of a socket. The readable bytes would be used as the constant buffer sequence for `send`, and the writable bytes used as the mutable buffer sequence for `receive`. — *end note*] Data written to the writable bytes of a dynamic buffer object is appended to the readable bytes of the same object.

2 A type `X` meets the `DynamicBuffer` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `MoveConstructible` (C++Std [moveconstructible]), as well as the additional requirements listed in Table 14.

3 In Table 14, `x` denotes a value of type `X`, `x1` denotes a (possibly const) value of type `X`, and `n` denotes a (possibly const) value of type `size_t`.

Table 14 — DynamicBuffer requirements

expression	type	assertion/note pre/post-conditions
`X::const_buffers_type`	type meeting ConstBufferSequence (16.2.2) requirements.	This type represents the memory associated with the readable bytes.
`X::mutable_buffers_type`	type meeting MutableBufferSequence (16.2.2) requirements.	This type represents the memory associated with the writable bytes.
`x1.size()`	`size_t`	Returns the number of readable bytes.
`x1.max_size()`	`size_t`	Returns the maximum number of bytes, both readable and writable, that can be held by `x1`.

Table 14 — DynamicBuffer requirements (continued)

expression	type	assertion/note pre/post-conditions
`x1.capacity()`	`size_t`	Returns the maximum number of bytes, both readable and writable, that can be held by `x1` without requiring reallocation.
`x1.data()`	`X::const_-buffers_type`	Returns a constant buffer sequence `u` that represents the readable bytes, and where `buffer_size(u) == size()`.
`x.prepare(n)`	`X::mutable_-buffers_type`	Returns a mutable buffer sequence `u` representing the writable bytes, and where `buffer_size(u) == n`. The dynamic buffer reallocates memory as required. All constant or mutable buffer sequences previously obtained using `data()` or `prepare()` are invalidated. *Throws:* `length_error` if `size() + n` exceeds `max_size()`.
`x.commit(n)`		Appends `n` bytes from the start of the writable bytes to the end of the readable bytes. The remainder of the writable bytes are discarded. If `n` is greater than the number of writable bytes, all writable bytes are appended to the readable bytes. All constant or mutable buffer sequences previously obtained using `data()` or `prepare()` are invalidated.
`x.consume(n)`		Removes `n` bytes from beginning of the readable bytes. If `n` is greater than the number of readable bytes, all readable bytes are removed. All constant or mutable buffer sequences previously obtained using `data()` or `prepare()` are invalidated.

16.2.4 Requirements on read and write operations [buffer.reqmts.read.write]

1 A *read operation* is an operation that reads data into a mutable buffer sequence argument of a type meeting `MutableBufferSequence` (16.2.1) requirements. The mutable buffer sequence specifies memory where the data should be placed. A read operation shall always fill a buffer in the sequence completely before proceeding to the next.

2 A *write operation* is an operation that writes data from a constant buffer sequence argument of a type meeting `ConstBufferSequence` (16.2.2) requirements. The constant buffer sequence specifies memory where the data to be written is located. A write operation shall always write a buffer in the sequence completely before proceeding to the next.

3 If a read or write operation is also an asynchronous operation (13.2.7), the operation shall maintain one or more copies of the buffer sequence until such time as the operation no longer requires access to the memory specified by the buffers in the sequence. The program shall ensure the memory remains valid until:

(3.1) — the last copy of the buffer sequence is destroyed, or

(3.2) — the completion handler for the asynchronous operation is invoked,

whichever comes first.

16.3 Error codes [buffer.err]

`const error_category& stream_category() noexcept;`

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

2 The object's `default_error_condition` and `equivalent` virtual functions behave as specified for the class `error_category`. The object's `name` virtual function returns a pointer to the string `"stream"`.

`error_code make_error_code(stream_errc e) noexcept;`

3 *Returns:* `error_code(static_cast<int>(e), stream_category())`.

`error_condition make_error_condition(stream_errc e) noexcept;`

4 *Returns:* `error_condition(static_cast<int>(e), stream_category())`.

16.4 Class `mutable_buffer` [buffer.mutable]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class mutable_buffer
  {
  public:
    // constructors:
    mutable_buffer() noexcept;
    mutable_buffer(void* p, size_t n) noexcept;

    // members:
    void* data() const noexcept;
    size_t size() const noexcept;
    mutable_buffer& operator+=(size_t n) noexcept;

  private:
    void* data_; // exposition only
    size_t size_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 The `mutable_buffer` class satisfies requirements of `MutableBufferSequence` (16.2.1), `DefaultConstructible` (C++Std [defaultconstructible]), and `CopyAssignable` (C++Std [copyassignable]).

`mutable_buffer() noexcept;`

2 *Postconditions:* `data_ == nullptr` and `size_ == 0`.

`mutable_buffer(void* p, size_t n) noexcept;`

3 *Postconditions:* `data_ == p` and `size_ == n`.

`void* data() const noexcept;`

4 *Returns:* `data_`.

```
size_t size() const noexcept;
```

5 *Returns:* `size_`.

```
mutable_buffer& operator+=(size_t n) noexcept;
```

6 *Effects:* Sets `data_` to `static_cast<char*>(data_) + min(n, size_)`, and then `size_` to `size_ - min(n, size_)`.

7 *Returns:* `*this`.

16.5 Class `const_buffer` [buffer.const]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class const_buffer
  {
  public:
    // constructors:
    const_buffer() noexcept;
    const_buffer(const void* p, size_t n) noexcept;
    const_buffer(const mutable_buffer& b) noexcept;

    // members:
    const void* data() const noexcept;
    size_t size() const noexcept;
    const_buffer& operator+=(size_t n) noexcept;

  private:
    const void* data_; // exposition only
    size_t size_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 The `const_buffer` class satisfies requirements of `ConstBufferSequence` (16.2.2), `DefaultConstructible` (C++Std [defaultconstructible]), and `CopyAssignable` (C++Std [copyassignable]).

```
const_buffer() noexcept;
```

2 *Postconditions:* `data_ == nullptr` and `size_ == 0`.

```
const_buffer(const void* p, size_t n) noexcept;
```

3 *Postconditions:* `data_ == p` and `size_ == n`.

```
const_buffer(const mutable_buffer& b);
```

4 *Postconditions:* `data_ == b.data()` and `size_ == b.size()`.

```
const void* data() const noexcept;
```

5 *Returns:* `data_`.

```
size_t size() const noexcept;
```

6 *Returns:* `size_`.

```
const_buffer& operator+=(size_t n) noexcept;
```

7 *Effects:* Sets `data_` to `static_cast<const char*>(data_) + min(n, size_)`, and then `size_` to `size_ - min(n, size_)`.

8 *Returns:* `*this`.

16.6 Buffer type traits [buffer.traits]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class T> struct is_mutable_buffer_sequence;
  template<class T> struct is_const_buffer_sequence;
  template<class T> struct is_dynamic_buffer;

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 This sub-clause contains templates that may be used to query the properties of a type at compile time. Each of these templates is a UnaryTypeTrait (C++Std [meta.rqmts]) with a BaseCharacteristic of `true_type` if the corresponding condition is true, otherwise `false_type`.

Table 15 — Buffer type traits

Template	Condition	Preconditions
`template<class T> struct is_mutable_buffer_sequence`	`T` meets the syntactic requirements for mutable buffer sequence (16.2.1).	`T` is a complete type.
`template<class T> struct is_const_buffer_sequence`	`T` meets the syntactic requirements for constant buffer sequence (16.2.2).	`T` is a complete type.
`template<class T> struct is_dynamic_buffer`	`T` meets the syntactic requirements for dynamic buffer (16.2.3).	`T` is a complete type.

16.7 Buffer sequence access [buffer.seq.access]

```
const mutable_buffer* buffer_sequence_begin(const mutable_buffer& b);
```

```
const const_buffer* buffer_sequence_begin(const const_buffer& b);
```

1 *Returns:* `std::addressof(b)`.

```
const mutable_buffer* buffer_sequence_end(const mutable_buffer& b);
const const_buffer* buffer_sequence_end(const const_buffer& b);
```

2 *Returns:* `std::addressof(b) + 1`.

```
template<class C> auto buffer_sequence_begin(C& c) -> decltype(c.begin());
template<class C> auto buffer_sequence_begin(const C& c) -> decltype(c.begin());
```

3 *Returns:* `c.begin()`.

```
template<class C> auto buffer_sequence_end(C& c) -> decltype(c.end());
template<class C> auto buffer_sequence_end(const C& c) -> decltype(c.end());
```

4 *Returns:* `c.end()`.

16.8 Function `buffer_size` [buffer.size]

```
template<class ConstBufferSequence>
  size_t buffer_size(const ConstBufferSequence& buffers) noexcept;
```

1 *Returns:* The total size of all buffers in the sequence, as if computed as follows:

```
size_t total_size = 0;
auto i = std::experimental::net::buffer_sequence_begin(buffers);
auto end = std::experimental::net::buffer_sequence_end(buffers);
for (; i != end; ++i)
{
  const_buffer b(*i);
  total_size += b.size();
}
return total_size;
```

16.9 Function `buffer_copy` [buffer.copy]

```
template<class MutableBufferSequence, class ConstBufferSequence>
  size_t buffer_copy(const MutableBufferSequence& dest,
                     const ConstBufferSequence& source) noexcept;
template<class MutableBufferSequence, class ConstBufferSequence>
  size_t buffer_copy(const MutableBufferSequence& dest,
                     const ConstBufferSequence& source,
                     size_t max_size) noexcept;
```

1 *Effects:* Copies bytes from the buffer sequence `source` to the buffer sequence `dest`, as if by calls to `memcpy`.

2 The number of bytes copied is the lesser of:

(2.1) — `buffer_size(dest)`;

(2.2) — `buffer_size(source)`; and

(2.3) — `max_size`, if specified.

3 The mutable buffer sequence `dest` specifies memory where the data should be placed. The operation always fills a buffer in the sequence completely before proceeding to the next.

4 The constant buffer sequence `source` specifies memory where the data to be written is located. The operation always copies a buffer in the sequence completely before proceeding to the next.

5 *Returns:* The number of bytes copied from `source` to `dest`.

16.10 Buffer arithmetic [buffer.arithmetic]

```
mutable_buffer operator+(const mutable_buffer& b, size_t n) noexcept;
mutable_buffer operator+(size_t n, const mutable_buffer& b) noexcept;
```

1 *Returns:* A `mutable_buffer` equivalent to

```
mutable_buffer(
  static_cast<char*>(b.data()) + min(n, b.size()),
  b.size() - min(n, b.size()));
```

```
const_buffer operator+(const const_buffer& b, size_t n) noexcept;
const_buffer operator+(size_t n, const const_buffer& b) noexcept;
```

2 *Returns:* A `const_buffer` equivalent to

```
const_buffer(
  static_cast<const char*>(b.data()) + min(n, b.size()),
  b.size() - min(n, b.size()));
```

16.11 Buffer creation functions [buffer.creation]

1 In the functions below, `T` must be a trivially copyable or standard-layout type (C++Std [basic.types]).

2 For the function overloads below that accept an argument of type `vector<>`, the buffer objects returned are invalidated by any vector operation that also invalidates all references, pointers and iterators referring to the elements in the sequence (C++Std [vector]).

3 For the function overloads below that accept an argument of type `basic_string<>`, the buffer objects returned are invalidated according to the rules defined for invalidation of references, pointers and iterators referring to elements of the sequence (C++Std [string.require]).

```
mutable_buffer buffer(void* p, size_t n) noexcept;
```

4 *Returns:* `mutable_buffer(p, n)`.

```
const_buffer buffer(const void* p, size_t n) noexcept;
```

5 *Returns:* `const_buffer(p, n)`.

```
mutable_buffer buffer(const mutable_buffer& b) noexcept;
```

6 *Returns:* `b`.

```
mutable_buffer buffer(const mutable_buffer& b, size_t n) noexcept;
```

7 *Returns:* `mutable_buffer(b.data(), min(b.size(), n))`.

```
const_buffer buffer(const const_buffer& b) noexcept;
```

8 *Returns:* `b`.

```
const_buffer buffer(const const_buffer& b, size_t n) noexcept;
```

9 *Returns:* `const_buffer(b.data(), min(b.size(), n))`.

```
template<class T, size_t N>
  mutable_buffer buffer(T (&data)[N]) noexcept;
template<class T, size_t N>
  const_buffer buffer(const T (&data)[N]) noexcept;
template<class T, size_t N>
```

```
  mutable_buffer buffer(array<T, N>& data) noexcept;
template<class T, size_t N>
  const_buffer buffer(array<const T, N>& data) noexcept;
template<class T, size_t N>
  const_buffer buffer(const array<T, N>& data) noexcept;
template<class T, class Allocator>
  mutable_buffer buffer(vector<T, Allocator>& data) noexcept;
template<class T, class Allocator>
  const_buffer buffer(const vector<T, Allocator>& data) noexcept;
template<class CharT, class Traits, class Allocator>
  mutable_buffer buffer(basic_string<CharT, Traits, Allocator>& data) noexcept;
template<class CharT, class Traits, class Allocator>
  const_buffer buffer(const basic_string<CharT, Traits, Allocator>& data) noexcept;
template<class CharT, class Traits>
  const_buffer buffer(basic_string_view<CharT, Traits> data) noexcept;
```

10 *Returns:*

```
buffer(
  begin(data) != end(data) ? std::addressof(*begin(data)) : nullptr,
  (end(data) - begin(data)) * sizeof(*begin(data)));
```

```
template<class T, size_t N>
  mutable_buffer buffer(T (&data)[N], size_t n) noexcept;
template<class T, size_t N>
  const_buffer buffer(const T (&data)[N], size_t n) noexcept;
template<class T, size_t N>
  mutable_buffer buffer(array<T, N>& data, size_t n) noexcept;
template<class T, size_t N>
  const_buffer buffer(array<const T, N>& data, size_t n) noexcept;
template<class T, size_t N>
  const_buffer buffer(const array<T, N>& data, size_t n) noexcept;
template<class T, class Allocator>
  mutable_buffer buffer(vector<T, Allocator>& data, size_t n) noexcept;
template<class T, class Allocator>
  const_buffer buffer(const vector<T, Allocator>& data, size_t n) noexcept;
template<class CharT, class Traits, class Allocator>
  mutable_buffer buffer(basic_string<CharT, Traits, Allocator>& data,
                        size_t n) noexcept;
template<class CharT, class Traits, class Allocator>
  const_buffer buffer(const basic_string<CharT, Traits, Allocator>& data,
                      size_t n) noexcept;
template<class CharT, class Traits>
  const_buffer buffer(basic_string_view<CharT, Traits> data,
                      size_t n) noexcept;
```

11 *Returns:* `buffer(buffer(data), n)`.

16.12 Class template `dynamic_vector_buffer` [buffer.dynamic.vector]

1 Class template `dynamic_vector_buffer` is an adaptor used to automatically grow or shrink a `vector` object, to reflect the data successfully transferred in an I/O operation.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
```

```
template<class T, class Allocator>
class dynamic_vector_buffer
{
public:
  // types:
  using const_buffers_type = const_buffer;
  using mutable_buffers_type = mutable_buffer;

  // constructors:
  explicit dynamic_vector_buffer(vector<T, Allocator>& vec) noexcept;
  dynamic_vector_buffer(vector<T, Allocator>& vec,
                        size_t maximum_size) noexcept;
  dynamic_vector_buffer(dynamic_vector_buffer&&) = default;

  // members:
  size_t size() const noexcept;
  size_t max_size() const noexcept;
  size_t capacity() const noexcept;
  const_buffers_type data() const noexcept;
  mutable_buffers_type prepare(size_t n);
  void commit(size_t n);
  void consume(size_t n);

private:
  vector<T, Allocator>& vec_; // exposition only
  size_t size_; // exposition only
  const size_t max_size_; // exposition only
};

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The `dynamic_vector_buffer` class template meets the requirements of `DynamicBuffer` (16.2.3).

3 The `dynamic_vector_buffer` class template requires that `T` is a trivially copyable or standard-layout type (C++Std [basic.types]) and that `sizeof(T) == 1`.

```
explicit dynamic_vector_buffer(vector<T, Allocator>& vec) noexcept;
```

4 *Effects:* Initializes `vec_` with `vec`, `size_` with `vec.size()`, and `max_size_` with `vec.max_size()`.

```
dynamic_vector_buffer(vector<T, Allocator>& vec,
                      size_t maximum_size) noexcept;
```

5 *Requires:* `vec.size() <= maximum_size`.

6 *Effects:* Initializes `vec_` with `vec`, `size_` with `vec.size()`, and `max_size_` with `maximum_size`.

```
size_t size() const noexcept;
```

7 *Returns:* `size_`.

```
size_t max_size() const noexcept;
```

8 *Returns:* `max_size_`.

```
size_t capacity() const noexcept;
```

9 *Returns:* `vec_.capacity()`.

```
const_buffers_type data() const noexcept;
```

10 *Returns:* `buffer(vec_, size_)`.

```
mutable_buffers_type prepare(size_t n);
```

11 *Effects:* Performs `vec_.resize(size_ + n)`.

12 *Returns:* `buffer(buffer(vec_) + size_, n)`.

13 *Remarks:* `length_error` if `size() + n` exceeds `max_size()`.

```
void commit(size_t n);
```

14 *Effects:* Performs:

```
size_ += min(n, vec_.size() - size_);
vec_.resize(size_);
```

```
void consume(size_t n);
```

15 *Effects:* Performs:

```
size_t m = min(n, size_);
vec_.erase(vec_.begin(), vec_.begin() + m);
size_ -= m;
```

16.13 Class template `dynamic_string_buffer` [buffer.dynamic.string]

1 Class template `dynamic_string_buffer` is an adaptor used to automatically grow or shrink a `basic_string` object, to reflect the data successfully transferred in an I/O operation.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class CharT, class Traits, class Allocator>
  class dynamic_string_buffer
  {
  public:
    // types:
    using const_buffers_type = const_buffer;
    using mutable_buffers_type = mutable_buffer;

    // constructors:
    explicit dynamic_string_buffer(basic_string<CharT, Traits, Allocator>& str) noexcept;
    dynamic_string_buffer(basic_string<CharT, Traits, Allocator>& str,
                          size_t maximum_size) noexcept;
    dynamic_string_buffer(dynamic_string_buffer&&) = default;

    // members:
    size_t size() const noexcept;
    size_t max_size() const noexcept;
    size_t capacity() const noexcept;
    const_buffers_type data() const noexcept;
```

```
    mutable_buffers_type prepare(size_t n);
    void commit(size_t n) noexcept;
    void consume(size_t n);

  private:
    basic_string<CharT, Traits, Allocator>& str_; // exposition only
    size_t size_; // exposition only
    const size_t max_size_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The `dynamic_string_buffer` class template meets the requirements of `DynamicBuffer` (16.2.3).

3 The `dynamic_string_buffer` class template requires that `sizeof(CharT) == 1`.

```
explicit dynamic_string_buffer(basic_string<CharT, Traits, Allocator>& str) noexcept;
```

4 *Effects:* Initializes `str_` with `str`, `size_` with `str.size()`, and `max_size_` with `str.max_size()`.

```
dynamic_string_buffer(basic_string<CharT, Traits, Allocator>& str,
                      size_t maximum_size) noexcept;
```

5 *Requires:* `str.size() <= maximum_size`.

6 *Effects:* Initializes `str_` with `str`, `size_` with `str.size()`, and `max_size_` with `maximum_size`.

```
size_t size() const noexcept;
```

7 *Returns:* `size_`.

```
size_t max_size() const noexcept;
```

8 *Returns:* `max_size_`.

```
size_t capacity() const noexcept;
```

9 *Returns:* `str_.capacity()`.

```
const_buffers_type data() const noexcept;
```

10 *Returns:* `buffer(str_, size_)`.

```
mutable_buffers_type prepare(size_t n);
```

11 *Effects:* Performs `str_.resize(size_ + n)`.

12 *Returns:* `buffer(buffer(str_) + size_, n)`.

13 *Remarks:* `length_error` if `size() + n` exceeds `max_size()`.

```
void commit(size_t n) noexcept;
```

14 *Effects:* Performs:

```
size_ += min(n, str_.size() - size_);
str_.resize(size_);
```

```
void consume(size_t n);
```

15 *Effects:* Performs:

```
size_t m = min(n, size_);
str_.erase(0, m);
size_ -= m;
```

16.14 Dynamic buffer creation functions [buffer.dynamic.creation]

```
template<class T, class Allocator>
  dynamic_vector_buffer<T, Allocator>
  dynamic_buffer(vector<T, Allocator>& vec) noexcept;
```

1 *Returns:* `dynamic_vector_buffer<T, Allocator>(vec)`.

```
template<class T, class Allocator>
  dynamic_vector_buffer<T, Allocator>
  dynamic_buffer(vector<T, Allocator>& vec, size_t n) noexcept;
```

2 *Returns:* `dynamic_vector_buffer<T, Allocator>(vec, n)`.

```
template<class CharT, class Traits, class Allocator>
  dynamic_string_buffer<CharT, Traits, Allocator>
  dynamic_buffer(basic_string<CharT, Traits, Allocator>& str) noexcept;
```

3 *Returns:* `dynamic_string_buffer<CharT, Traits, Allocator>(str)`.

```
template<class CharT, class Traits, class Allocator>
  dynamic_string_buffer<CharT, Traits, Allocator>
  dynamic_buffer(basic_string<CharT, Traits, Allocator>& str, size_t n) noexcept;
```

4 *Returns:* `dynamic_string_buffer<CharT, Traits, Allocator>(str, n)`.

17 Buffer-oriented streams [buffer.stream]

17.1 Requirements [buffer.stream.reqmts]

17.1.1 Buffer-oriented synchronous read stream requirements [buffer.stream.reqmts.syncreadstream]

1 A type `X` meets the `SyncReadStream` requirements if it satisfies the requirements listed in Table 16.

2 In Table 16, `a` denotes a value of type `X`, `mb` denotes a (possibly const) value satisfying the `MutableBufferSequence` (16.2.1) requirements, and `ec` denotes an object of type `error_code`.

Table 16 — SyncReadStream requirements

operation	type	semantics, pre/post-conditions
`a.read_some(mb)` `a.read_some(mb,ec)`	`size_t`	Meets the requirements for a read operation (16.2.4). If `buffer_size(mb) > 0`, reads one or more bytes of data from the stream `a` into the buffer sequence `mb`. If successful, sets `ec` such that `!ec` is `true`, and returns the number of bytes read. If an error occurred, sets `ec` such that `!!ec` is `true`, and returns 0. If all data has been read from the stream, and the stream performed an orderly shutdown, sets `ec` to `stream_errc::eof` and returns 0. If `buffer_size(mb) == 0`, the operation shall not block. Sets `ec` such that `!ec` is `true`, and returns 0.

17.1.2 Buffer-oriented asynchronous read stream requirements [buffer.stream.reqmts.asyncreadstream]

1 A type `X` meets the `AsyncReadStream` requirements if it satisfies the requirements listed below.

2 In the table below, `a` denotes a value of type `X`, `mb` denotes a (possibly const) value satisfying the `MutableBufferSequence` (16.2.1) requirements, and `t` is a completion token.

Table 17 — AsyncReadStream requirements

operation	type	semantics, pre/post-conditions
`a.get_executor()`	A type satisfying the Executor requirements (13.2.2).	Returns the associated I/O executor.

Table 17 — AsyncReadStream requirements (continued)

operation	type	semantics, pre/post-conditions
`a.async_read_some(mb,t)`	The return type is determined according to the requirements for an asynchronous operation (13.2.7).	Meets the requirements for a read operation (16.2.4) and an asynchronous operation (13.2.7) with completion signature `void(error_code ec, size_t n)`. If `buffer_size(mb) > 0`, initiates an asynchronous operation to read one or more bytes of data from the stream `a` into the buffer sequence `mb`. If successful, `ec` is set such that `!ec` is `true`, and `n` is the number of bytes read. If an error occurred, `ec` is set such that `!!ec` is `true`, and `n` is 0. If all data has been read from the stream, and the stream performed an orderly shutdown, `ec` is `stream_errc::eof` and `n` is 0. If `buffer_size(mb) == 0`, the operation completes immediately. `ec` is set such that `!ec` is `true`, and `n` is 0.

17.1.3 Buffer-oriented synchronous write stream requirements [buffer.stream.reqmts.syncwritestream]

1 A type `X` meets the `SyncWriteStream` requirements if it satisfies the requirements listed below.

2 In the table below, `a` denotes a value of type `X`, `cb` denotes a (possibly const) value satisfying the `ConstBufferSequence` (16.2.2) requirements, and `ec` denotes an object of type `error_code`.

Table 18 — SyncWriteStream requirements

operation	type	semantics, pre/post-conditions
`a.write_some(cb)` `a.write_some(cb,ec)`	`size_t`	Meets the requirements for a write operation (16.2.4). If `buffer_size(cb) > 0`, writes one or more bytes of data to the stream `a` from the buffer sequence `cb`. If successful, sets `ec` such that `!ec` is `true`, and returns the number of bytes written. If an error occurred, sets `ec` such that `!!ec` is `true`, and returns 0. If `buffer_size(cb) == 0`, the operation shall not block. Sets `ec` such that `!ec` is `true`, and returns 0.

17.1.4 Buffer-oriented asynchronous write stream requirements [buffer.stream.reqmts.asyncwritestream]

1 A type `X` meets the `AsyncWriteStream` requirements if it satisfies the requirements listed below.

2 In the table below, `a` denotes a value of type `X`, `cb` denotes a (possibly const) value satisfying the `ConstBufferSequence` (16.2.2) requirements, and `t` is a completion token.

Table 19 — AsyncWriteStream requirements

operation	type	semantics, pre/post-conditions
`a.get_executor()`	A type satisfying the Executor requirements (13.2.2).	Returns the associated I/O executor.
`a.async_write_some(cb,t)`	The return type is determined according to the requirements for an asynchronous operation (13.2.7).	Meets the requirements for a write operation (16.2.4) and an asynchronous operation (13.2.7) with completion signature `void(error_code ec, size_t n)`. If `buffer_size(cb) > 0`, initiates an asynchronous operation to write one or more bytes of data to the stream `a` from the buffer sequence `cb`. If successful, `ec` is set such that `!ec` is `true`, and `n` is the number of bytes written. If an error occurred, `ec` is set such that `!!ec` is `true`, and `n` is 0. If `buffer_size(cb) == 0`, the operation completes immediately. `ec` is set such that `!ec` is `true`, and `n` is 0.

17.1.5 Completion condition requirements [buffer.stream.reqmts.completioncondition]

1 A completion condition is a function object that is used with the algorithms `read` (17.5), `async_read` (17.6), `write` (17.7), and `async_write` (17.8) to determine when the algorithm has completed transferring data.

2 A type `X` meets the `CompletionCondition` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `CopyConstructible` (C++Std [copyconstructible]), as well as the additional requirements listed below.

3 In the table below, `x` denotes a value of type `X`, `ec` denotes a (possibly const) value of type `error_code`, and `n` denotes a (possibly const) value of type `size_t`.

Table 20 — CompletionCondition requirements

expression	return type	assertion/note pre/post-condition
`x(ec, n)`	`size_t`	Let `n` be the total number of bytes transferred by the read or write algorithm so far. Returns the maximum number of bytes to be transferred on the next `read_some`, `async_read_some`, `write_some`, or `async_write_some` operation performed by the algorithm. Returns `0` to indicate that the algorithm is complete.

17.2 Class `transfer_all` [buffer.stream.transfer.all]

1 The class `transfer_all` is a completion condition that is used to specify that a read or write operation should continue until all of the data has been transferred, or until an error occurs.

```
namespace std {
```

```
namespace experimental {
namespace net {
inline namespace v1 {

  class transfer_all
  {
  public:
    size_t operator()(const error_code& ec, size_t) const;
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The class `transfer_all` satisfies the `CompletionCondition` (17.1.5) requirements.

`size_t operator()(const error_code& ec, size_t) const;`

3 *Returns:* If `!ec`, an unspecified non-zero value. Otherwise `0`.

17.3 Class `transfer_at_least` [buffer.stream.transfer.at.least]

1 The class `transfer_at_least` is a completion condition that is used to specify that a read or write operation should continue until a minimum number of bytes has been transferred, or until an error occurs.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class transfer_at_least
  {
  public:
    explicit transfer_at_least(size_t m);
    size_t operator()(const error_code& ec, size_t n) const;
  private:
    size_t minimum_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The class `transfer_at_least` satisfies the `CompletionCondition` (17.1.5) requirements.

`explicit transfer_at_least(size_t m);`

3 *Postconditions:* `minimum_ == m`.

`size_t operator()(const error_code& ec, size_t n) const;`

4 *Returns:* If `!ec && n < minimum_`, an unspecified non-zero value. Otherwise `0`.

17.4 Class `transfer_exactly` [buffer.stream.transfer.exactly]

1 The class `transfer_exactly` is a completion condition that is used to specify that a read or write operation should continue until an exact number of bytes has been transferred, or until an error occurs.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class transfer_exactly
  {
  public:
    explicit transfer_exactly(size_t e);
    size_t operator()(const error_code& ec, size_t n) const;
  private:
    size_t exact_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The class `transfer_exactly` satisfies the `CompletionCondition` (17.1.5) requirements.

```
explicit transfer_exactly(size_t e);
```

3 *Postconditions:* `exact_ == e`.

```
size_t operator()(const error_code& ec, size_t n) const;
```

4 *Returns:* If `!ec && n < exact_`, the result of `min(exact_ - n, N)`, where `N` is an unspecified non-zero value. Otherwise `0`.

17.5 Synchronous read operations [buffer.read]

```
template<class SyncReadStream, class MutableBufferSequence>
  size_t read(SyncReadStream& stream,
              const MutableBufferSequence& buffers);
template<class SyncReadStream, class MutableBufferSequence>
  size_t read(SyncReadStream& stream,
              const MutableBufferSequence& buffers, error_code& ec);
template<class SyncReadStream, class MutableBufferSequence,
  class CompletionCondition>
    size_t read(SyncReadStream& stream,
                const MutableBufferSequence& buffers,
                CompletionCondition completion_condition);
template<class SyncReadStream, class MutableBufferSequence,
  class CompletionCondition>
    size_t read(SyncReadStream& stream,
                const MutableBufferSequence& buffers,
                CompletionCondition completion_condition,
                error_code& ec);
```

1 A read operation (16.2.4).

2 *Effects:* Clears `ec`, then reads data from the buffer-oriented synchronous read stream (17.1.1) object `stream` by performing zero or more calls to the stream's `read_some` member function.

3 The `completion_condition` parameter specifies a completion condition to be called prior to each call to the stream's `read_some` member function. The completion condition is passed the `error_code` value from the most recent `read_some` call, and the total number of bytes transferred in the synchronous

read operation so far. The completion condition return value specifies the maximum number of bytes to be read on the subsequent `read_some` call. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

4 The synchronous read operation continues until:

(4.1) — the total number of bytes transferred is equal to `buffer_size(buffers)`; or

(4.2) — the completion condition returns `0`.

5 On return, `ec` contains the `error_code` value from the most recent `read_some` call.

6 *Returns:* The total number of bytes transferred in the synchronous read operation.

7 *Remarks:* This function shall not participate in overload resolution unless `is_mutable_buffer_sequence<MutableBufferSequence>::value` is `true`.

```
template<class SyncReadStream, class DynamicBuffer>
  size_t read(SyncReadStream& stream, DynamicBuffer&& b);
template<class SyncReadStream, class DynamicBuffer>
  size_t read(SyncReadStream& stream, DynamicBuffer&& b, error_code& ec);
template<class SyncReadStream, class DynamicBuffer,
  class CompletionCondition>
    size_t read(SyncReadStream& stream, DynamicBuffer&& b,
                CompletionCondition completion_condition);
template<class SyncReadStream, class DynamicBuffer,
  class CompletionCondition>
    size_t read(SyncReadStream& stream, DynamicBuffer&& b,
                CompletionCondition completion_condition,
                error_code& ec);
```

8 *Effects:* Clears `ec`, then reads data from the synchronous read stream (17.1.1) object `stream` by performing zero or more calls to the stream's `read_some` member function.

9 Data is placed into the dynamic buffer (16.2.3) object `b`. A mutable buffer sequence (16.2.1) is obtained prior to each `read_some` call using `b.prepare(N)`, where `N` is an unspecified value less than or equal to `b.max_size() - b.size()`. [*Note:* Implementations are encouraged to use `b.capacity()` when determining `N`, to minimize the number of `read_some` calls performed on the stream. — *end note*] After each `read_some` call, the implementation performs `b.commit(n)`, where `n` is the return value from `read_some`.

10 The `completion_condition` parameter specifies a completion condition to be called prior to each call to the stream's `read_some` member function. The completion condition is passed the `error_code` value from the most recent `read_some` call, and the total number of bytes transferred in the synchronous read operation so far. The completion condition return value specifies the maximum number of bytes to be read on the subsequent `read_some` call. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

11 The synchronous read operation continues until:

(11.1) — `b.size() == b.max_size()`; or

(11.2) — the completion condition returns `0`.

12 On return, `ec` contains the `error_code` value from the most recent `read_some` call.

13 *Returns:* The total number of bytes transferred in the synchronous read operation.

14 *Remarks:* This function shall not participate in overload resolution unless `is_dynamic_buffer<DynamicBuffer>::value` is `true`.

17.6 Asynchronous read operations [buffer.async.read]

```
template<class AsyncReadStream, class MutableBufferSequence, class CompletionToken>
    DEDUCED async_read(AsyncReadStream& stream,
                       const MutableBufferSequence& buffers,
                       CompletionToken&& token);
template<class AsyncReadStream, class MutableBufferSequence, class CompletionCondition,
         class CompletionToken>
    DEDUCED async_read(AsyncReadStream& stream,
                       const MutableBufferSequence& buffers,
                       CompletionCondition completion_condition,
                       CompletionToken&& token);
```

1 A composed asynchronous read operation (13.2.7.14, 16.2.4).

2 *Completion signature:* `void(error_code ec, size_t n)`.

3 *Effects:* Reads data from the buffer-oriented asynchronous read stream (17.1.2) object `stream` by invoking the stream's `async_read_some` member function (henceforth referred to as asynchronous read_some operations) zero or more times.

4 The `completion_condition` parameter specifies a completion condition to be called prior to each asynchronous read_some operation. The completion condition is passed the `error_code` value from the most recent asynchronous read_some operation, and the total number of bytes transferred in the asynchronous read operation so far. The completion condition return value specifies the maximum number of bytes to be read on the subsequent asynchronous read_some operation. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

5 This asynchronous read operation is outstanding until:

(5.1) — the total number of bytes transferred is equal to `buffer_size(buffers)`; or

(5.2) — the completion condition returns `0`.

6 The program shall ensure the `AsyncReadStream` object `stream` is valid until the completion handler for the asynchronous operation is invoked.

7 On completion of the asynchronous operation, `ec` is the `error_code` value from the most recent asynchronous read_some operation, and `n` is the total number of bytes transferred.

8 *Remarks:* This function shall not participate in overload resolution unless `is_mutable_buffer_sequence<MutableBufferSequence>::value` is `true`.

```
template<class AsyncReadStream, class DynamicBuffer, class CompletionToken>
    DEDUCED async_read(AsyncReadStream& stream,
                       DynamicBuffer&& b, CompletionToken&& token);
template<class AsyncReadStream, class DynamicBuffer, class CompletionCondition,
         class CompletionToken>
    DEDUCED async_read(AsyncReadStream& stream,
                       DynamicBuffer&& b,
                       CompletionCondition completion_condition,
                       CompletionToken&& token);
```

9 *Completion signature:* `void(error_code ec, size_t n)`.

10 *Effects:* Initiates an asynchronous operation to read data from the buffer-oriented asynchronous read stream (17.1.2) object `stream` by performing one or more asynchronous read_some operations on the stream.

11 Data is placed into the dynamic buffer (16.2.3) object `b`. A mutable buffer sequence (16.2.1) is obtained prior to each `async_read_some` call using `b.prepare(N)`, where `N` is an unspecified value such that `N` is less than or equal to `b.max_size() - b.size()`. [*Note:* Implementations are encouraged to use

`b.capacity()` when determining `N`, to minimize the number of asynchronous read_some operations performed on the stream. — *end note*] After the completion of each asynchronous read_some operation, the implementation performs `b.commit(n)`, where `n` is the value passed to the asynchronous read_some operation's completion handler.

12 The `completion_condition` parameter specifies a completion condition to be called prior to each asynchronous read_some operation. The completion condition is passed the `error_code` value from the most recent asynchronous read_some operation, and the total number of bytes transferred in the asynchronous read operation so far. The completion condition return value specifies the maximum number of bytes to be read on the subsequent asynchronous read_some operation. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

13 The asynchronous read operation is outstanding until:

(13.1) — `b.size() == b.max_size()`; or

(13.2) — the completion condition returns `0`.

14 The program shall ensure the `AsyncReadStream` object `stream` is valid until the completion handler for the asynchronous operation is invoked.

15 On completion of the asynchronous operation, `ec` is the `error_code` value from the most recent asynchronous read_some operation, and `n` is the total number of bytes transferred.

16 *Remarks:* This function shall not participate in overload resolution unless `is_dynamic_buffer<DynamicBuffer>::value` is `true`.

17.7 Synchronous write operations [buffer.write]

```
template<class SyncWriteStream, class ConstBufferSequence>
  size_t write(SyncWriteStream& stream,
               const ConstBufferSequence& buffers);
template<class SyncWriteStream, class ConstBufferSequence>
  size_t write(SyncWriteStream& stream,
               const ConstBufferSequence& buffers, error_code& ec);
template<class SyncWriteStream, class ConstBufferSequence,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 const ConstBufferSequence& buffers,
                 CompletionCondition completion_condition);
template<class SyncWriteStream, class ConstBufferSequence,
  class CompletionCondition>
    size_t write(SyncWriteStream& stream,
                 const ConstBufferSequence& buffers,
                 CompletionCondition completion_condition,
                 error_code& ec);
```

1 A write operation (16.2.4).

2 *Effects:* Writes data to the buffer-oriented synchronous write stream (17.1.3) object `stream` by performing zero or more calls to the stream's `write_some` member function.

3 The `completion_condition` parameter specifies a completion condition to be called prior to each call to the stream's `write_some` member function. The completion condition is passed the `error_code` value from the most recent `write_some` call, and the total number of bytes transferred in the synchronous write operation so far. The completion condition return value specifies the maximum number of bytes to be written on the subsequent `write_some` call. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

4 The synchronous write operation continues until:

(4.1) — the total number of bytes transferred is equal to `buffer_size(buffers)`; or

(4.2) — the completion condition returns `0`.

5 On return, `ec` contains the `error_code` value from the most recent `write_some` call.

6 *Returns:* The total number of bytes transferred in the synchronous write operation.

7 *Remarks:* This function shall not participate in overload resolution unless `is_const_buffer_sequence<ConstBufferSequence>::value` is `true`.

```
template<class SyncWriteStream, class DynamicBuffer>
  size_t write(SyncWriteStream& stream, DynamicBuffer&& b);
template<class SyncWriteStream, class DynamicBuffer>
  size_t write(SyncWriteStream& stream, DynamicBuffer&& b, error_code& ec);
template<class SyncWriteStream, class DynamicBuffer, class CompletionCondition>
  size_t write(SyncWriteStream& stream, DynamicBuffer&& b,
               CompletionCondition completion_condition);
template<class SyncWriteStream, class DynamicBuffer, class CompletionCondition>
  size_t write(SyncWriteStream& stream, DynamicBuffer&& b,
               CompletionCondition completion_condition,
               error_code& ec);
```

8 *Effects:* Writes data to the synchronous write stream (17.1.3) object `stream` by performing zero or more calls to the stream's `write_some` member function.

9 Data is written from the dynamic buffer (16.2.3) object `b`. A constant buffer sequence (16.2.2) is obtained using `b.data()`. After the data has been written to the stream, the implementation performs `b.consume(n)`, where `n` is the number of bytes successfully written.

10 The `completion_condition` parameter specifies a completion condition to be called after each call to the stream's `write_some` member function. The completion condition is passed the `error_code` value from the most recent `write_some` call, and the total number of bytes transferred in the synchronous write operation so far. The completion condition return value specifies the maximum number of bytes to be written on the subsequent `write_some` call. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

11 The synchronous write operation continues until:

(11.1) — `b.size() == 0`; or

(11.2) — the completion condition returns `0`.

12 On return, `ec` contains the `error_code` value from the most recent `write_some` call.

13 *Returns:* The total number of bytes transferred in the synchronous write operation.

14 *Remarks:* This function shall not participate in overload resolution unless `is_dynamic_buffer<DynamicBuffer>::value` is `true`.

17.8 Asynchronous write operations [buffer.async.write]

```
template<class AsyncWriteStream, class ConstBufferSequence, class CompletionToken>
    DEDUCED async_write(AsyncWriteStream& stream,
                        const ConstBufferSequence& buffers,
                        CompletionToken&& token);
template<class AsyncWriteStream, class ConstBufferSequence, class CompletionCondition,
         class CompletionToken>
    DEDUCED async_write(AsyncWriteStream& stream,
                        const ConstBufferSequence& buffers,
                        CompletionCondition completion_condition,
                        CompletionToken&& token);
```

1 A composed asynchronous write operation (13.2.7.14, 16.2.4).

2 *Completion signature:* `void(error_code ec, size_t n)`.

3 *Effects:* Initiates an asynchronous operation to write data to the buffer-oriented asynchronous write stream (17.1.4) object `stream` by performing zero or more asynchronous operations on the stream using the stream's `async_write_some` member function (henceforth referred to as asynchronous write_some operations).

4 The `completion_condition` parameter specifies a completion condition to be called prior to each asynchronous write_some operation. The completion condition is passed the `error_code` value from the most recent asynchronous write_some operation, and the total number of bytes transferred in the asynchronous write operation so far. The completion condition return value specifies the maximum number of bytes to be written on the subsequent asynchronous write_some operation. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

5 The asynchronous write operation continues until:

(5.1) — the total number of bytes transferred is equal to `buffer_size(buffers)`; or

(5.2) — the completion condition returns `0`.

6 The program must ensure the `AsyncWriteStream` object `stream` is valid until the completion handler for the asynchronous operation is invoked.

7 On completion of the asynchronous operation, `ec` is the `error_code` value from the most recent asynchronous write_some operation, and `n` is the total number of bytes transferred.

8 *Remarks:* This function shall not participate in overload resolution unless `is_const_buffer_sequence<ConstBufferSequence>::value` is `true`.

```
template<class AsyncWriteStream, class DynamicBuffer, class CompletionToken>
  DEDUCED async_write(AsyncWriteStream& stream,
                      DynamicBuffer&& b, CompletionToken&& token);
template<class AsyncWriteStream, class DynamicBuffer, class CompletionCondition,
         class CompletionToken>
    DEDUCED async_write(AsyncWriteStream& stream,
                        DynamicBuffer&& b,
                        CompletionCondition completion_condition,
                        CompletionToken&& token);
```

9 *Completion signature:* `void(error_code ec, size_t n)`.

10 *Effects:* Initiates an asynchronous operation to write data to the buffer-oriented asynchronous write stream (17.1.4) object `stream` by performing zero or more asynchronous write_some operations on the stream.

11 Data is written from the dynamic buffer (16.2.3) object `b`. A constant buffer sequence (16.2.2) is obtained using `b.data()`. After the data has been written to the stream, the implementation performs `b.consume(n)`, where `n` is the number of bytes successfully written.

12 The `completion_condition` parameter specifies a completion condition to be called prior to each asynchronous write_some operation. The completion condition is passed the `error_code` value from the most recent asynchronous write_some operation, and the total number of bytes transferred in the asynchronous write operation so far. The completion condition return value specifies the maximum number of bytes to be written on the subsequent asynchronous write_some operation. Overloads where a completion condition is not specified behave as if called with an object of class `transfer_all`.

13 The asynchronous write operation continues until:

(13.1) — `b.size() == 0`; or

(13.2) — the completion condition returns `0`.

14 The program must ensure both the `AsyncWriteStream` object `stream` and the memory associated with the dynamic buffer `b` are valid until the completion handler for the asynchronous operation is invoked.

15 On completion of the asynchronous operation, `ec` is the `error_code` value from the most recent asynchronous write_some operation, and `n` is the total number of bytes transferred.

16 *Remarks:* This function shall not participate in overload resolution unless `is_dynamic_buffer<DynamicBuffer>::value` is `true`.

17.9 Synchronous delimited read operations [buffer.read.until]

```
template<class SyncReadStream, class DynamicBuffer>
  size_t read_until(SyncReadStream& s, DynamicBuffer&& b, char delim);
template<class SyncReadStream, class DynamicBuffer>
  size_t read_until(SyncReadStream& s, DynamicBuffer&& b,
                    char delim, error_code& ec);
template<class SyncReadStream, class DynamicBuffer>
  size_t read_until(SyncReadStream& s, DynamicBuffer&& b, string_view delim);
template<class SyncReadStream, class DynamicBuffer>
  size_t read_until(SyncReadStream& s, DynamicBuffer&& b,
                    string_view delim, error_code& ec);
```

1 *Effects:* Reads data from the buffer-oriented synchronous read stream (17.1.1) object `stream` by performing zero or more calls to the stream's `read_some` member function, until the input sequence of the dynamic buffer (16.2.3) object `b` contains the specified delimiter `delim`.

2 Data is placed into the dynamic buffer object `b`. A mutable buffer sequence (16.2.1) is obtained prior to each `read_some` call using `b.prepare(N)`, where `N` is an unspecified value such that `N <= max_size() - size()`. [*Note:* Implementations are encouraged to use `b.capacity()` when determining `N`, to minimize the number of `read_some` calls performed on the stream. — *end note*] After each `read_some` call, the implementation performs `b.commit(n)`, where `n` is the return value from `read_some`.

3 The synchronous read_until operation continues until:

(3.1) — the input sequence of `b` contains the delimiter `delim`; or

(3.2) — `b.size() == b.max_size()`; or

(3.3) — an asynchronous read_some operation fails.

4 On exit, if the input sequence of `b` contains the delimiter, `ec` is set such that `!ec` is `true`. Otherwise, if `b.size() == b.max_size()`, `ec` is set such that `ec == stream_errc::not_found`. If `b.size() < b.max_size()`, `ec` contains the `error_code` from the most recent `read_some` call.

5 *Returns:* The number of bytes in the input sequence of `b` up to and including the delimiter, if present. [*Note:* On completion, the buffer may contain additional bytes following the delimiter. — *end note*] Otherwise returns `0`.

17.10 Asynchronous delimited read operations [buffer.async.read.until]

```
template<class AsyncReadStream, class DynamicBuffer, class CompletionToken>
  DEDUCED async_read_until(AsyncReadStream& s,
                           DynamicBuffer&& b, char delim,
                           CompletionToken&& token);
template<class AsyncReadStream, class DynamicBuffer, class CompletionToken>
  DEDUCED async_read_until(AsyncReadStream& s,
                           DynamicBuffer&& b, string_view delim,
                           CompletionToken&& token);
```

1 A composed asynchronous operation (13.2.7.14).

2 *Completion signature:* `void(error_code ec, size_t n)`.

3 *Effects:* Initiates an asynchronous operation to read data from the buffer-oriented asynchronous read stream (17.1.2) object `stream` by performing zero or more asynchronous read_some operations on the stream, until the readable bytes of the dynamic buffer (16.2.3) object `b` contain the specified delimiter `delim`.

4 Data is placed into the dynamic buffer object `b`. A mutable buffer sequence (16.2.1) is obtained prior to each `async_read_some` call using `b.prepare(N)`, where `N` is an unspecified value such that `N <= max_size() - size()`. [*Note:* Implementations are encouraged to use `b.capacity()` when determining `N`, to minimize the number of asynchronous read_some operations performed on the stream. — *end note*] After the completion of each asynchronous read_some operation, the implementation performs `b.commit(n)`, where `n` is the value passed to the asynchronous read_some operation's completion handler.

5 The asynchronous read_until operation continues until:

(5.1) — the readable bytes of `b` contain the delimiter `delim`; or

(5.2) — `b.size() == b.max_size()`; or

(5.3) — an asynchronous read_some operation fails.

6 The program shall ensure the `AsyncReadStream` object `stream` is valid until the completion handler for the asynchronous operation is invoked.

7 If `delim` is of type `string_view`, the implementation copies the underlying sequence of characters prior to initiating an asynchronous read_some operation on the stream. [*Note:* This means that the caller is not required to guarantee the validity of the delimiter string after the call to `async_read_until` returns. — *end note*]

8 On completion of the asynchronous operation, if the readable bytes of `b` contain the delimiter, `ec` is set such that `!ec` is `true`. Otherwise, if `b.size() == b.max_size()`, `ec` is set such that `ec == stream_errc::not_found`. If `b.size() < b.max_size()`, `ec` is the `error_code` from the most recent asynchronous read_some operation. `n` is the number of readable bytes in `b` up to and including the delimiter, if present, otherwise `0`.

18 Sockets [socket]

18.1 Header <experimental/socket> synopsis [socket.synop]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  enum class socket_errc {
    already_open = an implementation defined non-zero value,
    not_found = an implementation defined non-zero value
  };

  const error_category& socket_category() noexcept;

  error_code make_error_code(socket_errc e) noexcept;
  error_condition make_error_condition(socket_errc e) noexcept;

  // Sockets:

  class socket_base;

  template<class Protocol>
    class basic_socket;

  template<class Protocol>
    class basic_datagram_socket;

  template<class Protocol>
    class basic_stream_socket;

  template<class Protocol>
    class basic_socket_acceptor;

  // Socket streams:

  template<class Protocol, class Clock = chrono::steady_clock,
    class WaitTraits = wait_traits<Clock>>
      class basic_socket_streambuf;

  template<class Protocol, class Clock = chrono::steady_clock,
    class WaitTraits = wait_traits<Clock>>
      class basic_socket_iostream;

  // synchronous connect operations:

  template<class Protocol, class EndpointSequence>
    typename Protocol::endpoint connect(basic_socket<Protocol>& s,
                                        const EndpointSequence& endpoints);
  template<class Protocol, class EndpointSequence>
    typename Protocol::endpoint connect(basic_socket<Protocol>& s,
```

```
                                        const EndpointSequence& endpoints,
                                        error_code& ec);
    template<class Protocol, class EndpointSequence, class ConnectCondition>
      typename Protocol::endpoint connect(basic_socket<Protocol>& s,
                                          const EndpointSequence& endpoints,
                                          ConnectCondition c);
    template<class Protocol, class EndpointSequence, class ConnectCondition>
      typename Protocol::endpoint connect(basic_socket<Protocol>& s,
                                          const EndpointSequence& endpoints,
                                          ConnectCondition c,
                                          error_code& ec);

    template<class Protocol, class InputIterator>
      InputIterator connect(basic_socket<Protocol>& s,
                            InputIterator first, InputIterator last);
    template<class Protocol, class InputIterator>
      InputIterator connect(basic_socket<Protocol>& s,
                            InputIterator first, InputIterator last,
                            error_code& ec);
    template<class Protocol, class InputIterator, class ConnectCondition>
      InputIterator connect(basic_socket<Protocol>& s,
                            InputIterator first, InputIterator last,
                            ConnectCondition c);
    template<class Protocol, class InputIterator, class ConnectCondition>
      InputIterator connect(basic_socket<Protocol>& s,
                            InputIterator first, InputIterator last,
                            ConnectCondition c,
                            error_code& ec);

    // asynchronous connect operations:

    template<class Protocol, class EndpointSequence, class CompletionToken>
      DEDUCED async_connect(basic_socket<Protocol>& s,
                            const EndpointSequence& endpoints,
                            CompletionToken&& token);
    template<class Protocol, class EndpointSequence, class ConnectCondition,
             class CompletionToken>
      DEDUCED async_connect(basic_socket<Protocol>& s,
                            const EndpointSequence& endpoints,
                            ConnectCondition c,
                            CompletionToken&& token);

    template<class Protocol, class InputIterator, class CompletionToken>
      DEDUCED async_connect(basic_socket<Protocol>& s,
                            InputIterator first, InputIterator last,
                            CompletionToken&& token);
    template<class Protocol, class InputIterator, class ConnectCondition,
             class CompletionToken>
      DEDUCED async_connect(basic_socket<Protocol>& s,
                            InputIterator first, InputIterator last,
                            ConnectCondition c,
                            CompletionToken&& token);

  } // inline namespace v1
  } // namespace net
```

```
} // namespace experimental

  template<> struct is_error_code_enum<
    experimental::net::v1::socket_errc>
      : public true_type {};

} // namespace std
```

1 The figure below illustrates relationships between various types described in this Technical Specification. A solid line from **A** to **B** that is terminated by an open arrow indicates that **A** is derived from **B**. A solid line from **A** to **B** that starts with a diamond and is terminated by a solid arrow indicates that **A** contains an object of type **B**. A dotted line from **A** to **B** indicates that **A** is a synonym for the class template **B** with the specified template argument.

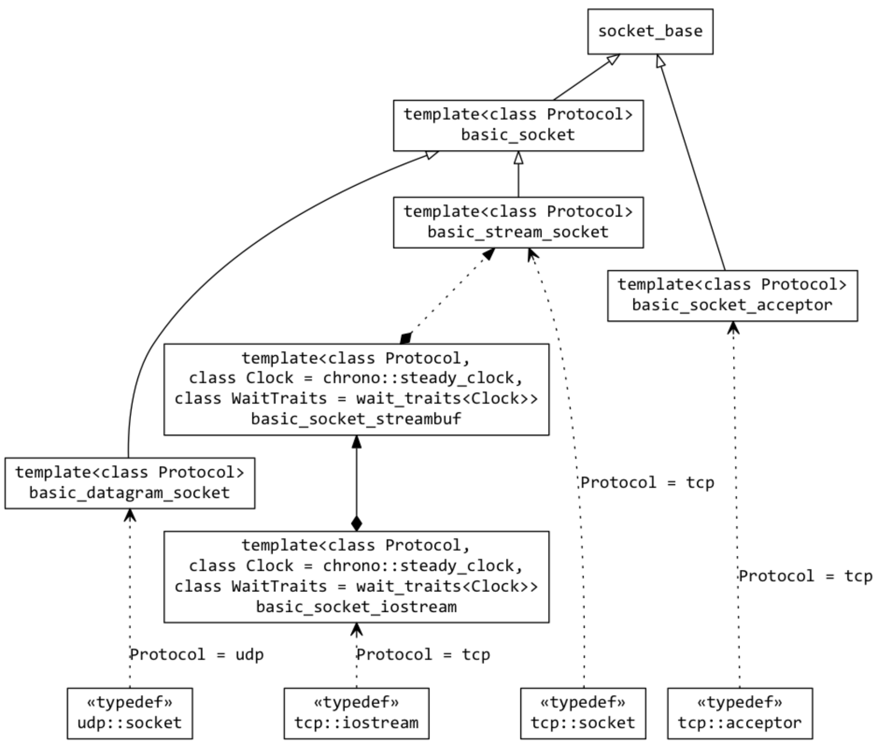

Figure 1 — Socket and socket stream types [non-normative]

18.2 Requirements [socket.reqmts]

18.2.1 Requirements on synchronous socket operations [socket.reqmts.sync]

1 In this section, *synchronous socket operations* are those member functions specified as two overloads, with and without an argument of type `error_code&`:

```
R f(A1 a1, A2 a2, ..., AN aN);
R f(A1 a1, A2 a2, ..., AN aN, error_code& ec);
```

2 For an object `s`, the conditions under which its synchronous socket operations may block the calling thread (C++Std [defns.block]) are determined as follows.

3 If:

(3.1) — `s.non_blocking() == true`,

(3.2) — the synchronous socket operation is specified in terms of a POSIX function other than `poll`,

(3.3) — that POSIX function lists `EWOULDBLOCK` or `EAGAIN` in its failure conditions, and

(3.4) — the effects of the operation cannot be established immediately

then the synchronous socket operation shall not block the calling thread. [*Note:* And the effects of the operation are not established. — *end note*]

4 Otherwise, the synchronous socket operation shall block the calling thread until the effects are established.

18.2.2 Requirements on asynchronous socket operations [socket.reqmts.async]

1 In this section, *asynchronous socket operations* are those member functions having prefix `async_`.

2 For an object `s`, a program may initiate asynchronous socket operations such that there are multiple simultaneously outstanding asynchronous operations.

3 When there are multiple outstanding asynchronous read operations (16.2.4) on `s`:

(3.1) — having no argument `flags` of type `socket_base::message_flags`, or

(3.2) — having an argument `flags` of type `socket_base::message_flags` but where `(flags & socket_base::message_out_of_band) == 0`

then the `buffers` are filled in the order in which these operations were issued. The order of invocation of the completion handlers for these operations is unspecified.

4 When there are multiple outstanding asynchronous read operations (16.2.4) on `s` having an argument `flags` of type `socket_base::message_flags` where `(flags & socket_base::message_out_of_band) != 0` then the `buffers` are filled in the order in which these operations were issued.

5 When there are multiple outstanding asynchronous write operations (16.2.4) on `s`, the `buffers` are transmitted in the order in which these operations were issued. The order of invocation of the completion handlers for these operations is unspecified.

18.2.3 Native handles [socket.reqmts.native]

1 Several classes described in this Technical Specification have a member type `native_handle_type`, a member function `native_handle`, and member functions that accept arguments of type `native_handle_type`. The presence of these members and their semantics is implementation-defined.

2 [*Note:* These members allow implementations to provide access to their implementation details. Their names are specified to facilitate portable compile-time detection. Actual use of these members is inherently non-portable. For operating systems that are based on POSIX, implementations are encouraged to define the `native_handle_type` for sockets as `int`, representing the native file descriptor associated with the socket. — *end note*]

18.2.4 Endpoint requirements [socket.reqmts.endpoint]

1 A type `X` meets the `Endpoint` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]), `DefaultConstructible` (C++Std [defaultconstructible]), `CopyConstructible` (C++Std [copyconstructible]), and `CopyAssignable` (C++Std [copyassignable]), as well as the additional requirements listed below.

2 In the table below, `a` denotes a (possibly const) value of type `X`, and `u` denotes an identifier.

Table 21 — Endpoint requirements

expression	type	assertion/note pre/post-conditions
`X::protocol_type`	type meeting `Protocol` (18.2.6) requirements	
`a.protocol()`	`protocol_type`	

3 In the table below, `a` denotes a (possibly const) value of type `X`, `b` denotes a value of type `X`, and `s` denotes a (possibly const) value of a type that is convertible to `size_t` and denotes a size in bytes.

Table 22 — Endpoint requirements for extensible implementations

expression	type	assertion/note pre/post-conditions
`a.data()`	`const void*`	Returns a pointer suitable for passing as the `address` argument to functions such as POSIX `connect`, or as the `dest_addr` argument to functions such as POSIX `sendto`. The implementation shall perform a `static_cast` on the pointer to convert it to `const sockaddr*`.
`b.data()`	`void*`	Returns a pointer suitable for passing as the `address` argument to functions such as POSIX `accept`, `getpeername`, `getsockname` and `recvfrom`. The implementation shall perform a `static_cast` on the pointer to convert it to `sockaddr*`.
`a.size()`	`size_t`	Returns a value suitable for passing as the `address_len` argument to functions such as POSIX `connect`, or as the `dest_len` argument to functions such as POSIX `sendto`, after appropriate integer conversion has been performed.
`b.resize(s)`		pre: `s >= 0` post: `a.size() == s` Passed the value contained in the `address_len` argument to functions such as POSIX `accept`, `getpeername`, `getsockname`, and `recvfrom`, after successful completion of the function. Permitted to throw an exception if the protocol associated with the endpoint object `a` does not support the specified size.
`a.capacity()`	`size_t`	Returns a value suitable for passing as the `address_len` argument to functions such as POSIX `accept`, `getpeername`, `getsockname`, and `recvfrom`, after appropriate integer conversion has been performed.

18.2.5 Endpoint sequence requirements [socket.reqmts.endpointsequence]

1 A type `X` meets the `EndpointSequence` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `CopyConstructible` (C++Std [copyconstructible]), as well as the additional requirements listed below.

2 In the table below, `x` denotes a (possibly const) value of type `X`.

Table 23 — EndpointSequence requirements

expression	return type	assertion/note pre/post-condition
`x.begin()` `x.end()`	A type meeting the requirements for forward iterators (C++Std [forward.iterators]) whose value type is convertible to a type satisfying the `Endpoint` (18.2.4) requirements.	`[x.begin(), x.end())` is a valid range.

18.2.6 Protocol requirements [socket.reqmts.protocol]

1 A type `X` meets the `Protocol` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]), `CopyConstructible` (C++Std [copyconstructible]), and `CopyAssignable` (C++Std [copyassignable]), as well as the additional requirements listed below.

Table 24 — Protocol requirements

expression	return type	assertion/note pre/post-conditions
`X::endpoint`	type meeting endpoint (18.2.4) requirements	

2 In the table below, `a` denotes a (possibly const) value of type `X`.

Table 25 — Protocol requirements for extensible implementations

expression	return type	assertion/note pre/post-conditions
`a.family()`	`int`	Returns a value suitable for passing as the domain argument to POSIX `socket` (or equivalent).
`a.type()`	`int`	Returns a value suitable for passing as the type argument to POSIX `socket` (or equivalent).
`a.protocol()`	`int`	Returns a value suitable for passing as the protocol argument to POSIX `socket` (or equivalent).

18.2.7 Acceptable protocol requirements [socket.reqmts.acceptableprotocol]

1 A type `X` meets the `AcceptableProtocol` requirements if it satisfies the requirements of `Protocol` (18.2.6) as well as the additional requirements listed below.

Table 26 — AcceptableProtocol requirements

expression	return type	assertion/note pre/post-conditions
`X::socket`	A type that satisfies the requirements of `Destructible` (C++Std [destructible]) and `MoveConstructible` (C++Std [moveconstructible]), and that is publicly and unambiguously derived from `basic_socket<X>`.	

18.2.8 Gettable socket option requirements [socket.reqmts.gettablesocketoption]

1 A type `X` meets the `GettableSocketOption` requirements if it satisfies the requirements listed below.

2 In the table below, `a` denotes a (possibly const) value of type `X`, `b` denotes a value of type `X`, `p` denotes a (possibly const) value that meets the `Protocol` (18.2.6) requirements, and `s` denotes a (possibly const) value of a type that is convertible to `size_t` and denotes a size in bytes.

Table 27 — GettableSocketOption requirements for extensible implementations

expression	type	assertion/note pre/post-conditions
`a.level(p)`	`int`	Returns a value suitable for passing as the `level` argument to POSIX `getsockopt` (or equivalent).
`a.name(p)`	`int`	Returns a value suitable for passing as the `option_name` argument to POSIX `getsockopt` (or equivalent).
`b.data(p)`	`void*`	Returns a pointer suitable for passing as the `option_value` argument to POSIX getsockopt (or equivalent).
`a.size(p)`	`size_t`	Returns a value suitable for passing as the `option_len` argument to POSIX `getsockopt` (or equivalent), after appropriate integer conversion has been performed.

Table 27 — GettableSocketOption requirements for extensible implementations (continued)

expression	type	assertion/note pre/post-conditions
`b.resize(p,s)`		post: `b.size(p) == s`. Passed the value contained in the `option_len` argument to POSIX `getsockopt` (or equivalent) after successful completion of the function. Permitted to throw an exception if the socket option object `b` does not support the specified size.

18.2.9 Settable socket option requirements [socket.reqmts.settablesocketoption]

1 A type `X` meets the `SettableSocketOption` requirements if it satisfies the requirements listed below.

2 In the table below, `a` denotes a (possibly const) value of type `X`, `p` denotes a (possibly const) value that meets the `Protocol` (18.2.6) requirements, and `u` denotes an identifier.

Table 28 — SettableSocketOption requirements for extensible implementations

expression	type	assertion/note pre/post-conditions
`a.level(p)`	`int`	Returns a value suitable for passing as the `level` argument to POSIX `setsockopt` (or equivalent).
`a.name(p)`	`int`	Returns a value suitable for passing as the `option_name` argument to POSIX `setsockopt` (or equivalent).
`a.data(p)`	`const void*`	Returns a pointer suitable for passing as the `option_value` argument to POSIX `setsockopt` (or equivalent).
`a.size(p)`	`size_t`	Returns a value suitable for passing as the `option_len` argument to POSIX `setsockopt` (or equivalent), after appropriate integer conversion has been performed.

18.2.10 Boolean socket options [socket.reqmts.opt.bool]

1 A type `X` meets the `BooleanSocketOption` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]), `DefaultConstructible` (C++Std [defaultconstructible]), `CopyConstructible` (C++Std [copyconstructible]), `CopyAssignable` (C++Std [copyassignable]), `GettableSocketOption` (18.2.8), and `SettableSocketOption` (18.2.9), `X` is contextually convertible to bool, and `X` satisfies the additional requirements listed below.

2 In the table below, `a` denotes a (possibly const) value of type `X`, `v` denotes a (possibly const) value of type `bool`, and `u` denotes an identifier.

Table 29 — BooleanSocketOption requirements

expression	type	assertion/note pre/post-conditions
`X u;`		post: `!u.value()`.
`X u(v);`		post: `u.value() == v`.
`a.value()`	`bool`	Returns the current boolean value of the socket option object.
`static_cast<bool>(a)`	`bool`	Returns `a.value()`.
`!a`	`bool`	Returns `!a.value()`.

3 In this Technical Specification, types that satisfy the `BooleanSocketOption` requirements are defined as follows.

```
class C
{
public:
  // constructors:
  C() noexcept;
  explicit C(bool v) noexcept;

  // members:
  C& operator=(bool v) noexcept;

  bool value() const noexcept;

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

4 Extensible implementations provide the following member functions:

```
class C
{
public:
  template<class Protocol> int level(const Protocol& p) const noexcept;
  template<class Protocol> int name(const Protocol& p) const noexcept;
  template<class Protocol> void* data(const Protocol& p) noexcept;
  template<class Protocol> const void* data(const Protocol& p) const noexcept;
  template<class Protocol> size_t size(const Protocol& p) const noexcept;
  template<class Protocol> void resize(const Protocol& p, size_t s);
  // remainder unchanged
private:
  int value_; // exposition only
};
```

5 Let *L* and *N* identify the POSIX macros to be passed as the `level` and `option_name` arguments, respectively, to POSIX `setsockopt` and `getsockopt`.

```
C() noexcept;
```

6 *Postconditions:* `!value()`.

```
explicit C(bool v) noexcept;
```

7 *Postconditions:* `value() == v`.

`C& operator=(bool v) noexcept;`

8 *Returns:* `*this`.

9 *Postconditions:* `value() == v`.

`bool value() const noexcept;`

10 *Returns:* The stored socket option value. For extensible implementations, returns `value_ != 0`.

`explicit operator bool() const noexcept;`

11 *Returns:* `value()`.

`bool operator!() const noexcept;`

12 *Returns:* `!value()`.

`template<class Protocol> int level(const Protocol& p) const noexcept;`

13 *Returns:* *L*.

`template<class Protocol> int name(const Protocol& p) const noexcept;`

14 *Returns:* *N*.

`template<class Protocol> void* data(const Protocol& p) noexcept;`

15 *Returns:* `std::addressof(value_)`.

`template<class Protocol> const void* data(const Protocol& p) const noexcept;`

16 *Returns:* `std::addressof(value_)`.

`template<class Protocol> size_t size(const Protocol& p) const noexcept;`

17 *Returns:* `sizeof(value_)`.

`template<class Protocol> void resize(const Protocol& p, size_t s);`

18 *Remarks:* `length_error` if `s` is not a valid data size for the protocol specified by `p`.

18.2.11 Integer socket options [socket.reqmts.opt.int]

1 A type `X` meets the `IntegerSocketOption` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]), `DefaultConstructible` (C++Std [defaultconstructible]), `CopyConstructible` (C++Std [copyconstructible]), `CopyAssignable` (C++Std [copyassignable]), `GettableSocketOption` (18.2.8), and `SettableSocketOption` (18.2.9), as well as the additional requirements listed below.

2 In the table below, `a` denotes a (possibly const) value of type `X`, `v` denotes a (possibly const) value of type `int`, and `u` denotes an identifier.

Table 30 — IntegerSocketOption requirements

expression	type	assertion/note pre/post-conditions
`X u;`		post: `u.value() == 0`.
`X u(v);`		post: `u.value() == v`.
`a.value()`	`int`	Returns the current integer value of the socket option object.

3 In this Technical Specification, types that satisfy the `IntegerSocketOption` requirements are defined as follows.

```
class C
{
public:
  // constructors:
  C() noexcept;
  explicit C(int v) noexcept;

  // members:
  C& operator=(int v) noexcept;

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

4 Extensible implementations provide the following member functions:

```
class C
{
public:
  template<class Protocol> int level(const Protocol& p) const noexcept;
  template<class Protocol> int name(const Protocol& p) const noexcept;
  template<class Protocol> void* data(const Protocol& p) noexcept;
  template<class Protocol> const void* data(const Protocol& p) const noexcept;
  template<class Protocol> size_t size(const Protocol& p) const noexcept;
  template<class Protocol> void resize(const Protocol& p, size_t s);
  // remainder unchanged
private:
  int value_; // exposition only
};
```

5 Let *L* and *N* identify the POSIX macros to be passed as the `level` and `option_name` arguments, respectively, to POSIX `setsockopt` and `getsockopt`.

`C() noexcept;`

6 *Postconditions:* `!value()`.

`explicit C(int v) noexcept;`

7 *Postconditions:* `value() == v`.

`C& operator=(int v) noexcept;`

8 *Returns:* `*this`.

9 *Postconditions:* `value() == v`.

`int value() const noexcept;`

10 *Returns:* The stored socket option value. For extensible implementations, returns `value_`.

`template<class Protocol> int level(const Protocol& p) const noexcept;`

11 *Returns:* *L*.

`template<class Protocol> int name(const Protocol& p) const noexcept;`

12 *Returns:* *N*.

```
template<class Protocol> void* data(const Protocol& p) noexcept;
```

13 *Returns:* `std::addressof(value_)`.

```
template<class Protocol> const void* data(const Protocol& p) const noexcept;
```

14 *Returns:* `std::addressof(value_)`.

```
template<class Protocol> size_t size(const Protocol& p) const noexcept;
```

15 *Returns:* `sizeof(value_)`.

```
template<class Protocol> void resize(const Protocol& p, size_t s);
```

16 *Remarks:* `length_error` if `s` is not a valid data size for the protocol specified by `p`.

18.2.12 I/O control command requirements [socket.reqmts.iocontrolcommand]

1 A type `X` meets the `IoControlCommand` requirements if it satisfies the requirements listed below.

2 In the table below, `a` denotes a (possibly const) value of type `X`, and `b` denotes a value of type `X`.

Table 31 — IoControlCommand requirements for extensible implementations

expression	type	assertion/note pre/post-conditions
`a.name()`	`int`	Returns a value suitable for passing as the request argument to POSIX `ioctl` (or equivalent).
`b.data()`	`void*`	

18.2.13 Connect condition requirements [socket.reqmts.connectcondition]

1 A type `X` meets the `ConnectCondition` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]) and `CopyConstructible` (C++Std [copyconstructible]), as well as the additional requirements listed below.

2 In the table below, `x` denotes a value of type `X`, `ec` denotes a (possibly const) value of type `error_code`, and `ep` denotes a (possibly const) value of some type satisfying the endpoint (18.2.4) requirements.

Table 32 — ConnectCondition requirements

expression	return type	assertion/note pre/post-condition
`x(ec, ep)`	`bool`	Returns `true` to indicate that the `connect` or `async_connect` algorithm should attempt a connection to the endpoint `ep`. Otherwise, returns `false` to indicate that the algorithm should not attempt connection to the endpoint `ep`, and should instead skip to the next endpoint in the sequence.

18.3 Error codes [socket.err]

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

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

2 The object's `default_error_condition` and `equivalent` virtual functions behave as specified for the class `error_category`. The object's `name` virtual function returns a pointer to the string `"socket"`.

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

3 *Returns:* `error_code(static_cast<int>(e), socket_category())`.

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

4 *Returns:* `error_condition(static_cast<int>(e), socket_category())`.

18.4 Class `socket_base` [socket.base]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class socket_base
  {
  public:
    class broadcast;
    class debug;
    class do_not_route;
    class keep_alive;
    class linger;
    class out_of_band_inline;
    class receive_buffer_size;
    class receive_low_watermark;
    class reuse_address;
    class send_buffer_size;
    class send_low_watermark;

    using shutdown_type = T1;
    static constexpr shutdown_type shutdown_receive;
    static constexpr shutdown_type shutdown_send;
    static constexpr shutdown_type shutdown_both;

    using wait_type = T2;
    static constexpr wait_type wait_read;
    static constexpr wait_type wait_write;
    static constexpr wait_type wait_error;

    using message_flags = T3;
    static constexpr message_flags message_peek;
    static constexpr message_flags message_out_of_band;
    static constexpr message_flags message_do_not_route;

    static const int max_listen_connections;

  protected:
    socket_base();
    ~socket_base();
  };
```

```
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 `socket_base` defines several member types:

(1.1) — socket option classes `broadcast`, `debug`, `do_not_route`, `keep_alive`, `linger`, `out_of_band_inline`, `receive_buffer_size`, `receive_low_watermark`, `reuse_address`, `send_buffer_size`, and `send_low_watermark`;

(1.2) — an enumerated type, `shutdown_type`, for use with the `basic_socket<Protocol>` class's `shutdown` member function.

(1.3) — an enumerated type, `wait_type`, for use with the `basic_socket<Protocol>` and `basic_socket_acceptor<Protocol>` classes' `wait` and `async_wait` member functions,

(1.4) — a bitmask type, `message_flags`, for use with the `basic_stream_socket<Protocol>` class's `send`, `async_send`, `receive`, and `async_receive` member functions, and the `basic_datagram_socket<Protocol>` class's `send`, `async_send`, `send_to`, `async_send_to`, `receive`, `async_receive`, `receive_from`, and `async_receive_from` member functions.

(1.5) — a constant, `max_listen_connections`, for use with the `basic_socket_acceptor<Protocol>` class's `listen` member function.

Table 33 — `socket_base` constants

Constant Name	POSIX macro	Definition or notes
`shutdown_receive`	`SHUT_RD`	Disables further receive operations.
`shutdown_send`	`SHUT_WR`	Disables further send operations.
`shutdown_both`	`SHUT_RDWR`	Disables further send and receive operations.
`wait_read`		Wait until the socket is ready-to-read. For a given socket, when a `wait` or `async_wait` operation using `wait_read` completes successfully, a subsequent call to the socket's `receive` or `receive_from` functions may complete without blocking. Similarly, for a given acceptor, when a `wait` or `async_wait` operation using `wait_read` completes successfully, a subsequent call to the acceptor's `accept` function may complete without blocking.
`wait_write`		Wait until the socket is ready-to-write. For a given socket, when a `wait` or `async_wait` operation using `wait_write` completes successfully, a subsequent call to the socket's `send` or `send_to` functions may complete without blocking.

Table 33 — `socket_base` constants (continued)

Constant Name	POSIX macro	Definition or notes
`wait_error`		Wait until the socket has a pending error condition. For a given socket, when a `wait` or `async_wait` operation using `wait_error` completes successfully, a subsequent call to one of the socket's synchronous operations may complete without blocking. The nature of the pending error condition determines which.
`message_peek`	`MSG_PEEK`	Leave received data in queue.
`message_out_of_band`	`MSG_OOB`	Out-of-band data.
`message_do_not_route`	`MSG_DONTROUTE`	Send without using routing tables.
`max_listen_connections`	`SOMAXCONN`	The implementation-defined limit on the length of the queue of pending incoming connections.

18.5 Socket options [socket.opt]

1 In the table below, let *C* denote a socket option class; let *L* identify the POSIX macro to be passed as the `level` argument to POSIX `setsockopt` and `getsockopt`; let *N* identify the POSIX macro to be passed as the `option_name` argument to POSIX `setsockopt` and `getsockopt`; and let *T* identify the type of the value whose address will be passed as the `option_value` argument to POSIX `setsockopt` and `getsockopt`.

Table 34 — Socket options

C	*L*	*N*	*T*	Requirements, definition or notes
`socket_base::broadcast`	`SOL_SOCKET`	`SO_BROADCAST`	`int`	Satisfies the `BooleanSocketOption` (18.2.10) type requirements. Determines whether a socket permits sending of broadcast messages, if supported by the protocol.
`socket_base::debug`	`SOL_SOCKET`	`SO_DEBUG`	`int`	Satisfies the `BooleanSocketOption` (18.2.10) type requirements. Determines whether debugging information is recorded by the underlying protocol.
`socket_base::do_not_route`	`SOL_SOCKET`	`SO_DONTROUTE`	`int`	Satisfies the `BooleanSocketOption` (18.2.10) type requirements. Determines whether outgoing messages bypass standard routing facilities.
`socket_base::keep_alive`	`SOL_SOCKET`	`SO_KEEPALIVE`	`int`	Satisfies the `BooleanSocketOption` (18.2.10) type requirements. Determines whether a socket permits sending of keep_alive messages, if supported by the protocol.
`socket_base::linger` (18.5.1)	`SOL_SOCKET`	`SO_LINGER`	`linger`	Controls the behavior when a socket is closed and unsent data is present.

Table 34 — Socket options (continued)

C	*L*	*N*	*T*	**Requirements, definition or notes**
`socket_base::out_of_band_inline`	`SOL_SOCKET`	`SO_OOBINLINE`	`int`	Satisfies the `BooleanSocketOption` (18.2.10) type requirements. Determines whether out-of-band data (also known as urgent data) is received inline.
`socket_base::receive_buffer_size`	`SOL_SOCKET`	`SO_RCVBUF`	`int`	Satisfies the `IntegerSocketOption` (18.2.11) type requirements. Specifies the size of the receive buffer associated with a socket.
`socket_base::receive_low_watermark`	`SOL_SOCKET`	`SO_RCVLOWAT`	`int`	Satisfies the `IntegerSocketOption` (18.2.11) type requirements. Specifies the minimum number of bytes to process for socket input operations.
`socket_base::reuse_address`	`SOL_SOCKET`	`SO_REUSEADDR`	`int`	Satisfies the `BooleanSocketOption` (18.2.10) type requirements. Determines whether the validation of endpoints used for binding a socket should allow the reuse of local endpoints, if supported by the protocol.
`socket_base::send_buffer_size`	`SOL_SOCKET`	`SO_SNDBUF`	`int`	Satisfies the `IntegerSocketOption` (18.2.11) type requirements. Specifies the size of the send buffer associated with a socket.
`socket_base::send_low_watermark`	`SOL_SOCKET`	`SO_SNDLOWAT`	`int`	Satisfies the `IntegerSocketOption` (18.2.11) type requirements. Specifies the minimum number of bytes to process for socket output operations.

18.5.1 Class `socket_base::linger` [socket.opt.linger]

1 The `linger` class represents a socket option for controlling the behavior when a socket is closed and unsent data is present.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class socket_base::linger
  {
  public:
    // constructors:
    linger() noexcept;
    linger(bool e, chrono::seconds t) noexcept;

    // members:
    bool enabled() const noexcept;
```

```
    void enabled(bool e) noexcept;

    chrono::seconds timeout() const noexcept;
    void timeout(chrono::seconds t) noexcept;
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 `linger` satisfies the requirements of `Destructible` (C++Std [destructible]), `DefaultConstructible` (C++Std [defaultconstructible]), `CopyConstructible` (C++Std [copyconstructible]), `CopyAssignable` (C++Std [copy-assignable]), `GettableSocketOption` (18.2.8), and `SettableSocketOption` (18.2.9).

3 Extensible implementations provide the following member functions:

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  class socket_base::linger
  {
  public:
    template<class Protocol> int level(const Protocol& p) const noexcept;
    template<class Protocol> int name(const Protocol& p) const noexcept;
    template<class Protocol> void data(const Protocol& p) noexcept;
    template<class Protocol> const void* data(const Protocol& p) const noexcept;
    template<class Protocol> size_t size(const Protocol& p) const noexcept;
    template<class Protocol> void resize(const Protocol& p, size_t s);
    // remainder unchanged
  private:
    ::linger value_;  // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

```
linger() noexcept;
```

4 *Postconditions:* `!enabled() && timeout() == chrono::seconds(0)`.

```
linger(bool e, chrono::seconds t) noexcept;
```

5 *Postconditions:* `enabled() == e && timeout() == t`.

```
bool enabled() const noexcept;
```

6 *Returns:* `value_.l_onoff != 0`.

```
void enabled(bool e) noexcept;
```

7 *Postconditions:* `enabled() == e`.

```
chrono::seconds timeout() const noexcept;
```

8 *Returns:* `chrono::seconds(value_.l_linger)`.

```
void timeout(chrono::seconds t) noexcept;
```

9 *Postconditions:* `timeout() == t`.

```
template<class Protocol> int level(const Protocol& p) const noexcept;
```

10 *Returns:* `SOL_SOCKET`.

```
template<class Protocol> int name(const Protocol& p) const noexcept;
```

11 *Returns:* `SO_LINGER`.

```
template<class Protocol> void* data(const Protocol& p) const noexcept;
```

12 *Returns:* `std::addressof(value_)`.

```
template<class Protocol> const void* data(const Protocol& p) const noexcept;
```

13 *Returns:* `std::addressof(value_)`.

```
template<class Protocol> size_t size(const Protocol& p) const noexcept;
```

14 *Returns:* `sizeof(value_)`.

```
template<class Protocol> void resize(const Protocol& p, size_t s);
```

15 *Remarks:* `length_error` if `s != sizeof(value_)`.

18.6 Class template basic_socket [socket.basic]

1 Class template `basic_socket<Protocol>` is used as the base class for the `basic_datagram_socket<Protocol>` and `basic_stream_socket<Protocol>` class templates. It provides functionality that is common to both types of socket.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Protocol>
  class basic_socket : public socket_base
  {
  public:
    // types:

    using executor_type = io_context::executor_type;
    using native_handle_type = implementation defined; // see 18.2.3
    using protocol_type = Protocol;
    using endpoint_type = typename protocol_type::endpoint;

    // basic_socket operations:

    executor_type get_executor() noexcept;

    native_handle_type native_handle(); // see 18.2.3

    void open(const protocol_type& protocol = protocol_type());
    void open(const protocol_type& protocol, error_code& ec);
```

```
void assign(const protocol_type& protocol,
            const native_handle_type& native_socket); // see 18.2.3
void assign(const protocol_type& protocol,
            const native_handle_type& native_socket,
            error_code& ec); // see 18.2.3

bool is_open() const noexcept;

void close();
void close(error_code& ec);

void cancel();
void cancel(error_code& ec);

template<class SettableSocketOption>
  void set_option(const SettableSocketOption& option);
template<class SettableSocketOption>
  void set_option(const SettableSocketOption& option, error_code& ec);

template<class GettableSocketOption>
  void get_option(GettableSocketOption& option) const;
template<class GettableSocketOption>
  void get_option(GettableSocketOption& option, error_code& ec) const;

template<class IoControlCommand>
  void io_control(IoControlCommand& command);
template<class IoControlCommand>
  void io_control(IoControlCommand& command, error_code& ec);

void non_blocking(bool mode);
void non_blocking(bool mode, error_code& ec);
bool non_blocking() const;

void native_non_blocking(bool mode);
void native_non_blocking(bool mode, error_code& ec);
bool native_non_blocking() const;

bool at_mark() const;
bool at_mark(error_code& ec) const;

size_t available() const;
size_t available(error_code& ec) const;

void bind(const endpoint_type& endpoint);
void bind(const endpoint_type& endpoint, error_code& ec);

void shutdown(shutdown_type what);
void shutdown(shutdown_type what, error_code& ec);

endpoint_type local_endpoint() const;
endpoint_type local_endpoint(error_code& ec) const;

endpoint_type remote_endpoint() const;
endpoint_type remote_endpoint(error_code& ec) const;
```

```
    void connect(const endpoint_type& endpoint);
    void connect(const endpoint_type& endpoint, error_code& ec);

    template<class CompletionToken>
      DEDUCED async_connect(const endpoint_type& endpoint,
                            CompletionToken&& token);

    void wait(wait_type w);
    void wait(wait_type w, error_code& ec);

    template<class CompletionToken>
      DEDUCED async_wait(wait_type w, CompletionToken&& token);

  protected:
    // construct / copy / destroy:

    explicit basic_socket(io_context& ctx);
    basic_socket(io_context& ctx, const protocol_type& protocol);
    basic_socket(io_context& ctx, const endpoint_type& endpoint);
    basic_socket(io_context& ctx, const protocol_type& protocol,
                 const native_handle_type& native_socket); // see 18.2.3
    basic_socket(const basic_socket&) = delete;
    basic_socket(basic_socket&& rhs);
    template<class OtherProtocol>
      basic_socket(basic_socket<OtherProtocol>&& rhs);

    ~basic_socket();

    basic_socket& operator=(const basic_socket&) = delete;
    basic_socket& operator=(basic_socket&& rhs);
    template<class OtherProtocol>
      basic_socket& operator=(basic_socket<OtherProtocol>&& rhs);

  private:
    protocol_type protocol_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Instances of class template `basic_socket` meet the requirements of `Destructible` (C++Std [destructible]), `MoveConstructible` (C++Std [moveconstructible]), and `MoveAssignable` (C++Std [moveassignable]).

3 When an operation has its effects specified as if by passing the result of `native_handle()` to a POSIX function, then the operation fails with error condition `errc::bad_file_descriptor` if `is_open() == false` at the point in the effects when the POSIX function is called.

18.6.1 `basic_socket` constructors [socket.basic.cons]

`explicit basic_socket(io_context& ctx);`

1 *Postconditions:*

(1.1) — `get_executor() == ctx.get_executor()`.

(1.2) — `is_open() == false`.

```
basic_socket(io_context& ctx, const protocol_type& protocol);
```

2 *Effects:* Opens this socket as if by calling `open(protocol)`.

3 *Postconditions:*

(3.1) — `get_executor() == ctx.get_executor()`.

(3.2) — `is_open() == true`.

(3.3) — `non_blocking() == false`.

(3.4) — `protocol_ == protocol`.

```
basic_socket(io_context& ctx, const endpoint_type& endpoint);
```

4 *Effects:* Opens and binds this socket as if by calling:

```
open(endpoint.protocol());
bind(endpoint);
```

5 *Postconditions:*

(5.1) — `get_executor() == ctx.get_executor()`.

(5.2) — `is_open() == true`.

(5.3) — `non_blocking() == false`.

(5.4) — `protocol_ == endpoint.protocol()`.

```
basic_socket(io_context& ctx, const protocol_type& protocol,
             const native_handle_type& native_socket);
```

6 *Requires:* `native_socket` is a native handle to an open socket.

7 *Effects:* Assigns the existing native socket into this socket as if by calling `assign(protocol, native_socket)`.

8 *Postconditions:*

(8.1) — `get_executor() == ctx.get_executor()`.

(8.2) — `is_open() == true`.

(8.3) — `non_blocking() == false`.

(8.4) — `protocol_ == protocol`.

```
basic_socket(basic_socket&& rhs);
```

9 *Effects:* Move constructs an object of class `basic_socket<Protocol>` that refers to the state originally represented by `rhs`.

10 *Postconditions:*

(10.1) — `get_executor() == rhs.get_executor()`.

(10.2) — `is_open()` returns the same value as `rhs.is_open()` prior to the constructor invocation.

(10.3) — `non_blocking()` returns the same value as `rhs.non_blocking()` prior to the constructor invocation.

(10.4) — `native_handle()` returns the prior value of `rhs.native_handle()`.

(10.5) — `protocol_` is the prior value of `rhs.protocol_`.

(10.6) — `rhs.is_open() == false`.

```
template<class OtherProtocol>
  basic_socket(basic_socket<OtherProtocol>&& rhs);
```

11 *Requires:* `OtherProtocol` is implicitly convertible to `Protocol`.

12 *Effects:* Move constructs an object of class `basic_socket<Protocol>` that refers to the state originally represented by `rhs`.

13 *Postconditions:*

(13.1) — `get_executor() == rhs.get_executor()`.

(13.2) — `is_open()` returns the same value as `rhs.is_open()` prior to the constructor invocation.

(13.3) — `non_blocking()` returns the same value as `rhs.non_blocking()` prior to the constructor invocation.

(13.4) — `native_handle()` returns the prior value of `rhs.native_handle()`.

(13.5) — `protocol_` is the result of converting the prior value of `rhs.protocol_`.

(13.6) — `rhs.is_open() == false`.

14 *Remarks:* This constructor shall not participate in overload resolution unless `OtherProtocol` is implicitly convertible to `Protocol`.

18.6.2 `basic_socket` destructor [socket.basic.dtor]

```
~basic_socket();
```

1 *Effects:* If `is_open()` is `true`, cancels all outstanding asynchronous operations associated with this socket, disables the linger socket option to prevent the destructor from blocking, and releases socket resources as if by POSIX `close(native_handle())`. Completion handlers for canceled operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

18.6.3 `basic_socket` assignment [socket.basic.assign]

```
basic_socket& operator=(basic_socket&& rhs);
```

1 *Effects:* If `is_open()` is `true`, cancels all outstanding asynchronous operations associated with this socket. Completion handlers for canceled operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`. Disables the linger socket option to prevent the assignment from blocking, and releases socket resources as if by POSIX `close(native_handle())`. Moves into `*this` the state originally represented by `rhs`.

2 *Postconditions:*

(2.1) — `get_executor() == rhs.get_executor()`.

(2.2) — `is_open()` returns the same value as `rhs.is_open()` prior to the assignment.

(2.3) — `non_blocking()` returns the same value as `rhs.non_blocking()` prior to the assignment.

(2.4) — `protocol_` is the prior value of `rhs.protocol_`.

(2.5) — `rhs.is_open() == false`.

3 *Returns:* `*this`.

```
template<class OtherProtocol>
  basic_socket& operator=(basic_socket<OtherProtocol>&& rhs);
```

4 *Requires:* `OtherProtocol` is implicitly convertible to `Protocol`.

5 *Effects:* If `is_open()` is `true`, cancels all outstanding asynchronous operations associated with this socket. Completion handlers for canceled operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`. Disables the linger socket option to prevent the assignment from blocking, and releases socket resources as if by POSIX `close(native_handle())`. Moves into `*this` the state originally represented by `rhs`.

6 *Postconditions:*

(6.1) — `get_executor() == rhs.get_executor()`.

(6.2) — `is_open()` returns the same value as `rhs.is_open()` prior to the assignment.

(6.3) — `non_blocking()` returns the same value as `rhs.non_blocking()` prior to the assignment.

(6.4) — `protocol_` is the result of converting the prior value of `rhs.protocol_`.

(6.5) — `rhs.is_open() == false`.

7 *Returns:* `*this`.

8 *Remarks:* This assignment operator shall not participate in overload resolution unless `OtherProtocol` is implicitly convertible to `Protocol`.

18.6.4 basic_socket operations [socket.basic.ops]

```
executor_type get_executor() noexcept;
```

1 *Returns:* The associated executor.

```
native_handle_type native_handle();
```

2 *Returns:* The native representation of this socket.

```
void open(const protocol_type& protocol);
void open(const protocol_type& protocol, error_code& ec);
```

3 *Effects:* Establishes the postcondition, as if by POSIX `socket(protocol.family(), protocol.type(), protocol.protocol())`.

4 *Postconditions:*

(4.1) — `is_open() == true`.

(4.2) — `non_blocking() == false`.

(4.3) — `protocol_ == protocol`.

5 *Error conditions:*

(5.1) — `socket_errc::already_open` — if `is_open() == true`.

```
void assign(const protocol_type& protocol,
            const native_handle_type& native_socket);
void assign(const protocol_type& protocol,
            const native_handle_type& native_socket, error_code& ec);
```

6 *Requires:* `native_socket` is a native handle to an open socket.

7 *Effects:* Assigns the native socket handle to this socket object.

8 *Postconditions:*

(8.1) — `is_open() == true`.

(8.2) — `non_blocking() == false`.

(8.3) — `protocol_ == protocol`.

9 *Error conditions:*

(9.1) — `socket_errc::already_open` — if `is_open() == true`.

```
bool is_open() const noexcept;
```

10 *Returns:* A `bool` indicating whether this socket was opened by a previous call to `open` or `assign`.

```
void close();
void close(error_code& ec);
```

11 *Effects:* If `is_open()` is `true`, cancels all outstanding asynchronous operations associated with this socket, and establishes the postcondition as if by POSIX `close(native_handle())`. Completion handlers for canceled asynchronous operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

12 *Postconditions:* `is_open() == false`.

```
void cancel();
void cancel(error_code& ec);
```

13 *Effects:* Cancels all outstanding asynchronous operations associated with this socket. Completion handlers for canceled asynchronous operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

14 *Error conditions:*

(14.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

15 *Remarks:* Does not block (C++Std [defns.block]) the calling thread pending completion of the canceled operations.

```
template<class SettableSocketOption>
  void set_option(const SettableSocketOption& option);
template<class SettableSocketOption>
  void set_option(const SettableSocketOption& option, error_code& ec);
```

16 *Effects:* Sets an option on this socket, as if by POSIX `setsockopt(native_handle(), option.level(protocol_), option.name(protocol_), option.data(protocol_), option.size(protocol_))`.

```
template<class GettableSocketOption>
  void get_option(GettableSocketOption& option);
template<class GettableSocketOption>
  void get_option(GettableSocketOption& option, error_code& ec);
```

17 *Effects:* Gets an option from this socket, as if by POSIX:

```
socklen_t option_len = option.size(protocol_);
int result = getsockopt(native_handle(), option.level(protocol_),
                        option.name(protocol_), option.data(protocol_),
                        &option_len);
if (result == 0)
  option.resize(option_len);
```

```
template<class IoControlCommand>
  void io_control(IoControlCommand& command);
template<class IoControlCommand>
  void io_control(IoControlCommand& command, error_code& ec);
```

18 *Effects:* Executes an I/O control command on this socket, as if by POSIX `ioctl(native_handle(), command.name(), command.data())`.

```
void non_blocking(bool mode);
void non_blocking(bool mode, error_code& ec);
```

19 *Effects:* Sets the non-blocking mode of this socket. The non-blocking mode determines whether subsequent synchronous socket operations (18.2.1) on `*this` block the calling thread.

20 *Error conditions:*

(20.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

21 *Postconditions:* `non_blocking() == mode`.

22 [*Note:* The non-blocking mode has no effect on the behavior of asynchronous operations. — *end note*]

```
bool non_blocking() const;
```

23 *Returns:* The non-blocking mode of this socket.

```
void native_non_blocking(bool mode);
void native_non_blocking(bool mode, error_code& ec);
```

24 *Effects:* Sets the non-blocking mode of the underlying native socket, as if by POSIX:

```
int flags = fcntl(native_handle(), F_GETFL, 0);
if (flags >= 0)
{
  if (mode)
    flags |= O_NONBLOCK;
  else
    flags &= ~O_NONBLOCK;
  fcntl(native_handle(), F_SETFL, flags);
}
```

25 The native non-blocking mode has no effect on the behavior of the synchronous or asynchronous operations specified in this clause.

26 *Error conditions:*

(26.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

(26.2) — `errc::invalid_argument` — if `mode == false` and `non_blocking() == true`. [*Note:* As the combination does not make sense. — *end note*]

```
bool native_non_blocking() const;
```

27 *Returns:* The non-blocking mode of the underlying native socket.

28 *Remarks:* Implementations are permitted and encouraged to cache the native non-blocking mode that was applied through a prior call to `native_non_blocking`. Implementations may return an incorrect value if a program sets the non-blocking mode directly on the socket, by calling an operating system-specific function on the result of `native_handle()`.

```
bool at_mark() const;
bool at_mark(error_code& ec) const;
```

29 *Effects:* Determines if this socket is at the out-of-band data mark, as if by POSIX `sockatmark(native_handle())`. [*Note:* The `at_mark()` function must be used in conjunction with the `socket_base::out_of_band_inline` socket option. — *end note*]

30 *Returns:* A `bool` indicating whether this socket is at the out-of-band data mark. `false` if an error occurs.

```
size_t available() const;
size_t available(error_code& ec) const;
```

31 *Returns:* An indication of the number of bytes that may be read without blocking, or `0` if an error occurs.

32 *Error conditions:*

(32.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

```
void bind(const endpoint_type& endpoint);
void bind(const endpoint_type& endpoint, error_code& ec);
```

33 *Effects:* Binds this socket to the specified local endpoint, as if by POSIX `bind(native_handle(), endpoint.data(), endpoint.size())`.

```
void shutdown(shutdown_type what);
void shutdown(shutdown_type what, error_code& ec);
```

34 *Effects:* Shuts down all or part of a full-duplex connection for the socket, as if by POSIX `shutdown(native_handle(), static_cast<int>(what))`.

```
endpoint_type local_endpoint() const;
endpoint_type local_endpoint(error_code& ec) const;
```

35 *Effects:* Determines the locally-bound endpoint associated with the socket, as if by POSIX:

```
endpoint_type endpoint;
socklen_t endpoint_len = endpoint.capacity();
int result = getsockname(native_handle(), endpoint.data(), &endpoint_len);
if (result == 0)
  endpoint.resize(endpoint_len);
```

36 *Returns:* On success, `endpoint`. Otherwise `endpoint_type()`.

```
endpoint_type remote_endpoint() const;
endpoint_type remote_endpoint(error_code& ec) const;
```

37 *Effects:* Determines the remote endpoint associated with this socket, as if by POSIX:

```
endpoint_type endpoint;
socklen_t endpoint_len = endpoint.capacity();
int result = getpeername(native_handle(), endpoint.data(), &endpoint_len);
if (result == 0)
  endpoint.resize(endpoint_len);
```

38 *Returns:* On success, `endpoint`. Otherwise `endpoint_type()`.

```
void connect(const endpoint_type& endpoint);
void connect(const endpoint_type& endpoint, error_code& ec);
```

39 *Effects:* If `is_open()` is `false`, opens this socket by performing `open(endpoint.protocol(), ec)`. If `ec`, returns with no further action. Connects this socket to the specified remote endpoint, as if by POSIX `connect(native_handle(), endpoint.data(), endpoint.size())`.

```
template<class CompletionToken>
  DEDUCED async_connect(const endpoint_type& endpoint, CompletionToken&& token);
```

40 *Completion signature:* `void(error_code ec)`.

41 *Effects:* If `is_open()` is `false`, opens this socket by performing `open(endpoint.protocol(), ec)`. If `ec`, the operation completes immediately with no further action. Initiates an asynchronous operation to connect this socket to the specified remote endpoint, as if by POSIX `connect(native_handle(), endpoint.data(), endpoint.size())`.

42 When an asynchronous connect operation on this socket is simultaneously outstanding with another asynchronous connect, read, or write operation on this socket, the behavior is undefined.

43 If a program performs a synchronous operation on this socket, other than `close` or `cancel`, while there is an outstanding asynchronous connect operation, the behavior is undefined.

```
void wait(wait_type w);
void wait(wait_type w, error_code& ec);
```

44 *Effects:* Waits for this socket to be ready to read, ready to write, or to have error conditions pending, as if by POSIX `poll`.

45 *Error conditions:*

(45.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

```
template<class CompletionToken>
  DEDUCED async_wait(wait_type w, CompletionToken&& token);
```

46 *Completion signature:* `void(error_code ec)`.

47 *Effects:* Initiates an asynchronous operation to wait for this socket to be ready to read, ready to write, or to have error conditions pending, as if by POSIX `poll`.

48 When there are multiple outstanding asynchronous wait operations on this socket with the same `wait_type` value, all of these operations complete when this socket enters the corresponding ready state. The order of invocation of the completion handlers for these operations is unspecified.

49 *Error conditions:*

(49.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

18.7 Class template `basic_datagram_socket` [socket.dgram]

1 The class template `basic_datagram_socket<Protocol>` is used to send and receive discrete messages of fixed maximum length.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Protocol>
  class basic_datagram_socket : public basic_socket<Protocol>
  {
  public:
```

```
// types:

using native_handle_type = implementation defined; // see 18.2.3
using protocol_type = Protocol;
using endpoint_type = typename protocol_type::endpoint;

// construct / copy / destroy:

explicit basic_datagram_socket(io_context& ctx);
basic_datagram_socket(io_context& ctx, const protocol_type& protocol);
basic_datagram_socket(io_context& ctx, const endpoint_type& endpoint);
basic_datagram_socket(io_context& ctx, const protocol_type& protocol,
                      const native_handle_type& native_socket);
basic_datagram_socket(const basic_datagram_socket&) = delete;
basic_datagram_socket(basic_datagram_socket&& rhs);
template<class OtherProtocol>
  basic_datagram_socket(basic_datagram_socket<OtherProtocol>&& rhs);

~basic_datagram_socket();

basic_datagram_socket& operator=(const basic_datagram_socket&) = delete;
basic_datagram_socket& operator=(basic_datagram_socket&& rhs);
template<class OtherProtocol>
  basic_datagram_socket& operator=(basic_datagram_socket<OtherProtocol>&& rhs);

// basic_datagram_socket operations:

template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 error_code& ec);

template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags, error_code& ec);

template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive(const MutableBufferSequence& buffers,
                        CompletionToken&& token);

template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive(const MutableBufferSequence& buffers,
                        socket_base::message_flags flags,
                        CompletionToken&& token);

template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender);
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender, error_code& ec);
```

```
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender,
                      socket_base::message_flags flags);
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender,
                      socket_base::message_flags flags,
                      error_code& ec);

template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive_from(const MutableBufferSequence& buffers,
                             endpoint_type& sender,
                             CompletionToken&& token);

template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive_from(const MutableBufferSequence& buffers,
                             endpoint_type& sender,
                             socket_base::message_flags flags,
                             CompletionToken&& token);

template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers, error_code& ec);

template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags, error_code& ec);

template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_send(const ConstBufferSequence& buffers,
                     CompletionToken&& token);

template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_send(const ConstBufferSequence& buffers,
                     socket_base::message_flags flags,
                     CompletionToken&& token);

template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& recipient);
template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& recipient, error_code& ec);

template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& recipient,
                 socket_base::message_flags flags);
template<class ConstBufferSequence>
```

```
    size_t send_to(const ConstBufferSequence& buffers,
                   const endpoint_type& recipient,
                   socket_base::message_flags flags, error_code& ec);

  template<class ConstBufferSequence, class CompletionToken>
    DEDUCED async_send_to(const ConstBufferSequence& buffers,
                          const endpoint_type& recipient,
                          CompletionToken&& token);

  template<class ConstBufferSequence, class CompletionToken>
    DEDUCED async_send_to(const ConstBufferSequence& buffers,
                          const endpoint_type& recipient,
                          socket_base::message_flags flags,
                          CompletionToken&& token);
};

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Instances of class template `basic_datagram_socket` meet the requirements of `Destructible` (C++Std [destructible]), `MoveConstructible` (C++Std [moveconstructible]), and `MoveAssignable` (C++Std [moveassignable]).

3 If a program performs a synchronous operation on this socket, other than `close`, `cancel`, `shutdown`, `send`, or `send_to`, while there is an outstanding asynchronous read operation, the behavior is undefined.

4 If a program performs a synchronous operation on this socket, other than `close`, `cancel`, `shutdown`, `receive`, or `receive_from`, while there is an outstanding asynchronous write operation, the behavior is undefined.

5 When an operation has its effects specified as if by passing the result of `native_handle()` to a POSIX function, then the operation fails with error condition `errc::bad_file_descriptor` if `is_open() == false` at the point in the effects when the POSIX function is called.

18.7.1 `basic_datagram_socket` constructors [socket.dgram.cons]

```
explicit basic_datagram_socket(io_context& ctx);
```

1 *Effects:* Initializes the base class with `basic_socket<Protocol>(ctx)`.

```
basic_datagram_socket(io_context& ctx, const protocol_type& protocol);
```

2 *Effects:* Initializes the base class with `basic_socket<Protocol>(ctx, protocol)`.

```
basic_datagram_socket(io_context& ctx, const endpoint_type& endpoint);
```

3 *Effects:* Initializes the base class with `basic_socket<Protocol>(ctx, endpoint)`.

```
basic_datagram_socket(io_context& ctx, const protocol_type& protocol,
                      const native_handle_type& native_socket);
```

4 *Effects:* Initializes the base class with `basic_socket<Protocol>(ctx, protocol, native_socket)`.

```
basic_datagram_socket(basic_datagram_socket&& rhs);
```

5 *Effects:* Move constructs an object of class `basic_datagram_socket<Protocol>`, initializing the base class with `basic_socket<Protocol>(std::move(rhs))`.

```
template<class OtherProtocol>
  basic_datagram_socket(basic_datagram_socket<OtherProtocol>&& rhs);
```

6 *Requires:* `OtherProtocol` is implicitly convertible to `Protocol`.

7 *Effects:* Move constructs an object of class `basic_datagram_socket<Protocol>`, initializing the base class with `basic_socket<Protocol>(std::move(rhs))`.

8 *Remarks:* This constructor shall not participate in overload resolution unless `OtherProtocol` is implicitly convertible to `Protocol`.

18.7.2 basic_datagram_socket assignment [socket.dgram.assign]

```
basic_datagram_socket& operator=(basic_datagram_socket&& rhs);
```

1 *Effects:* Equivalent to `basic_socket<Protocol>::operator=(std::move(rhs))`.

2 *Returns:* `*this`.

```
template<class OtherProtocol>
  basic_datagram_socket& operator=(basic_datagram_socket<OtherProtocol>&& rhs);
```

3 *Requires:* `OtherProtocol` is implicitly convertible to `Protocol`.

4 *Effects:* Equivalent to `basic_socket<Protocol>::operator=(std::move(rhs))`.

5 *Returns:* `*this`.

6 *Remarks:* This assignment operator shall not participate in overload resolution unless `OtherProtocol` is implicitly convertible to `Protocol`.

18.7.3 basic_datagram_socket operations [socket.dgram.op]

```
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 error_code& ec);
```

1 *Returns:* `receive(buffers, socket_base::message_flags(), ec)`.

```
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags, error_code& ec);
```

2 A read operation (16.2.4).

3 *Effects:* Constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, and reads data from this socket as if by POSIX:

```
msghdr message;
message.msg_name = nullptr;
message.msg_namelen = 0;
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
recvmsg(native_handle(), &message, static_cast<int>(flags));
```

4 *Returns:* On success, the number of bytes received. Otherwise `0`.

5 [*Note:* This operation may be used with connection-mode or connectionless-mode sockets, but it is normally used with connection-mode sockets because it does not permit the application to retrieve the source endpoint of received data. — *end note*]

```
template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive(const MutableBufferSequence& buffers,
                        CompletionToken&& token);
```

6 *Returns:* `async_receive(buffers, socket_base::message_flags(), std::forward<CompletionToken>(token))`.

```
template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive(const MutableBufferSequence& buffers,
                        socket_base::message_flags flags,
                        CompletionToken&& token);
```

7 *Completion signature:* `void(error_code ec, size_t n)`.

8 *Effects:* Initiates an asynchronous operation to read data from this socket. Constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, then reads data as if by POSIX:

```
msghdr message;
message.msg_name = nullptr;
message.msg_namelen = 0;
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
recvmsg(native_handle(), &message, static_cast<int>(flags));
```

9 If the operation completes successfully, `n` is the number of bytes received. Otherwise `n` is `0`.

10 [*Note:* This operation may be used with connection-mode or connectionless-mode sockets, but it is normally used with connection-mode sockets because it does not permit the application to retrieve the source endpoint of received data. — *end note*]

11 *Error conditions:*

(11.1) — `errc::invalid_argument` — if `socket_base::message_peek` is set in flags.

```
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender);
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender, error_code& ec);
```

12 *Returns:* `receive_from(buffers, sender, socket_base::message_flags(), ec)`.

```
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
                      endpoint_type& sender,
                      socket_base::message_flags flags);
template<class MutableBufferSequence>
  size_t receive_from(const MutableBufferSequence& buffers,
```

```
                        endpoint_type& sender,
                        socket_base::message_flags flags,
                        error_code& ec);
```

13 A read operation (16.2.4).

14 *Effects:* Constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, and reads data from this socket as if by POSIX:

```
msghdr message;
message.msg_name = sender.data();
message.msg_namelen = sender.capacity();
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
ssize_t result = recvmsg(native_handle(), &message, static_cast<int>(flags));
if (result >= 0)
  sender.resize(message.msg_namelen);
```

15 *Returns:* On success, the number of bytes received. Otherwise `0`.

16 [*Note:* This operation may be used with connection-mode or connectionless-mode sockets, but it is normally used with connectionless-mode sockets because it permits the application to retrieve the source endpoint of received data. — *end note*]

```
template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive_from(const MutableBufferSequence& buffers,
                             endpoint_type& sender,
                             CompletionToken&& token);
```

17 *Effects:* Returns `async_receive_from(buffers, sender, socket_base::message_flags(), forward<CompletionToken>(token))`.

```
template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive_from(const MutableBufferSequence& buffers,
                             endpoint_type& sender,
                             socket_base::message_flags flags,
                             CompletionToken&& token);
```

18 A read operation (16.2.4).

19 *Completion signature:* `void(error_code ec, size_t n)`.

20 *Effects:* Initiates an asynchronous operation to read data from this socket. Constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, then reads data as if by POSIX:

```
msghdr message;
message.msg_name = sender.data();
message.msg_namelen = sender.capacity();
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
ssize_t result = recvmsg(native_handle(), &message, static_cast<int>(flags));
if (result >= 0)
  sender.resize(message.msg_namelen);
```

21 If the operation completes successfully, `n` is the number of bytes received. Otherwise `n` is `0`.

22 [*Note:* This operation may be used with connection-mode or connectionless-mode sockets, but it is normally used with connectionless-mode sockets because it permits the application to retrieve the source endpoint of received data. — *end note*]

23 *Error conditions:*

(23.1) — `errc::invalid_argument` — if `socket_base::message_peek` is set in flags.

```
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers, error_code& ec);
```

24 *Returns:* `send(buffers, socket_base::message_flags(), ec)`.

```
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags, error_code& ec);
```

25 A write operation (16.2.4).

26 *Effects:* Constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, and writes data to this socket as if by POSIX:

```
msghdr message;
message.msg_name = nullptr;
message.msg_namelen = 0;
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
sendmsg(native_handle(), &message, static_cast<int>(flags));
```

27 *Returns:* On success, the number of bytes sent. Otherwise `0`.

```
template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_send(const ConstBufferSequence& buffers, CompletionToken&& token);
```

28 *Returns:* `async_send(buffers, socket_base::message_flags(), forward<CompletionToken>(token))`.

```
template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_send(const ConstBufferSequence& buffers,
                     socket_base::message_flags flags,
                     CompletionToken&& token);
```

29 A write operation (16.2.4).

30 *Completion signature:* `void(error_code ec, size_t n)`.

31 *Effects:* Initiates an asynchronous operation to write data to this socket. Constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, then writes data as if by POSIX:

```
msghdr message;
message.msg_name = nullptr;
message.msg_namelen = 0;
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
sendmsg(native_handle(), &message, static_cast<int>(flags));
```

32 If the operation completes successfully, `n` is the number of bytes sent. Otherwise `n` is `0`.

```
template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& recipient);
template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& recipient, error_code& ec);
```

33 *Returns:* `send_to(buffers, recipient, socket_base::message_flags(), ec)`.

```
template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& recipient,
                 socket_base::message_flags flags);
template<class ConstBufferSequence>
  size_t send_to(const ConstBufferSequence& buffers,
                 const endpoint_type& recipient,
                 socket_base::message_flags flags, error_code& ec);
```

34 A write operation (16.2.4).

35 *Effects:* Constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, and writes data to this socket as if by POSIX:

```
msghdr message;
message.msg_name = recipient.data();
message.msg_namelen = recipient.size();
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
sendmsg(native_handle(), &message, static_cast<int>(flags));
```

36 *Returns:* On success, the number of bytes sent. Otherwise `0`.

```
template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_send_to(const ConstBufferSequence& buffers,
                        const endpoint_type& recipient,
                        CompletionToken&& token);
```

37 *Returns:* `async_send_to(buffers, recipient, socket_base::message_flags(), forward<CompletionToken>(token))`.

```
template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_send_to(const ConstBufferSequence& buffers,
                        const endpoint_type& recipient,
```

```
                        socket_base::message_flags flags,
                        CompletionToken&& token);
```

38 A write operation (16.2.4).

39 *Completion signature:* `void(error_code ec, size_t n)`.

40 *Effects:* Initiates an asynchronous operation to write data to this socket. Constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, then writes data as if by POSIX:

```
msghdr message;
message.msg_name = recipient.data();
message.msg_namelen = recipient.size();
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
sendmsg(native_handle(), &message, static_cast<int>(flags));
```

41 If the operation completes successfully, `n` is the number of bytes sent. Otherwise `n` is `0`.

18.8 Class template `basic_stream_socket` [socket.stream]

1 The class template `basic_stream_socket<Protocol>` is used to exchange data with a peer over a sequenced, reliable, bidirectional, connection-mode byte stream.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Protocol>
  class basic_stream_socket : public basic_socket<Protocol>
  {
  public:
    // types:

    using native_handle_type = implementation defined; // see 18.2.3
    using protocol_type = Protocol;
    using endpoint_type = typename protocol_type::endpoint;

    // construct / copy / destroy:

    explicit basic_stream_socket(io_context& ctx);
    basic_stream_socket(io_context& ctx, const protocol_type& protocol);
    basic_stream_socket(io_context& ctx, const endpoint_type& endpoint);
    basic_stream_socket(io_context& ctx, const protocol_type& protocol,
                        const native_handle_type& native_socket);
    basic_stream_socket(const basic_stream_socket&) = delete;
    basic_stream_socket(basic_stream_socket&& rhs);
    template<class OtherProtocol>
      basic_stream_socket(basic_stream_socket<OtherProtocol>&& rhs);

    ~basic_stream_socket();

    basic_stream_socket& operator=(const basic_stream_socket&) = delete;
```

```
basic_stream_socket& operator=(basic_stream_socket&& rhs);
template<class OtherProtocol>
  basic_stream_socket& operator=(basic_stream_socket<OtherProtocol>&& rhs);

// basic_stream_socket operations:

template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 error_code& ec);

template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags, error_code& ec);

template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive(const MutableBufferSequence& buffers,
                        CompletionToken&& token);

template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive(const MutableBufferSequence& buffers,
                        socket_base::message_flags flags,
                        CompletionToken&& token);

template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers, error_code& ec);

template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags, error_code& ec);

template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_send(const ConstBufferSequence& buffers,
                     CompletionToken&& token);

template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_send(const ConstBufferSequence& buffers,
                     socket_base::message_flags flags,
                     CompletionToken&& token);

template<class MutableBufferSequence>
  size_t read_some(const MutableBufferSequence& buffers);
template<class MutableBufferSequence>
  size_t read_some(const MutableBufferSequence& buffers,
                   error_code& ec);
```

```
    template<class MutableBufferSequence, class CompletionToken>
      DEDUCED async_read_some(const MutableBufferSequence& buffers,
                              CompletionToken&& token);

    template<class ConstBufferSequence>
      size_t write_some(const ConstBufferSequence& buffers);
    template<class ConstBufferSequence>
      size_t write_some(const ConstBufferSequence& buffers,
                        error_code& ec);

    template<class ConstBufferSequence, class CompletionToken>
      DEDUCED async_write_some(const ConstBufferSequence& buffers,
                               CompletionToken&& token);
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Instances of class template `basic_stream_socket` meet the requirements of `Destructible` (C++Std [destructible]), `MoveConstructible` (C++Std [moveconstructible]), `MoveAssignable` (C++Std [moveassignable]), `SyncReadStream` (17.1.1), `SyncWriteStream` (17.1.3), `AsyncReadStream` (17.1.2), and `AsyncWriteStream` (17.1.4).

3 If a program performs a synchronous operation on this socket, other than `close`, `cancel`, `shutdown`, or `send`, while there is an outstanding asynchronous read operation, the behavior is undefined.

4 If a program performs a synchronous operation on this socket, other than `close`, `cancel`, `shutdown`, or `receive`, while there is an outstanding asynchronous write operation, the behavior is undefined.

5 When an operation has its effects specified as if by passing the result of `native_handle()` to a POSIX function, then the operation fails with error condition `errc::bad_file_descriptor` if `is_open() == false` at the point in the effects when the POSIX function is called.

18.8.1 `basic_stream_socket` constructors [socket.stream.cons]

```
explicit basic_stream_socket(io_context& ctx);
```

1 *Effects:* Initializes the base class with `basic_socket<Protocol>(ctx)`.

```
basic_stream_socket(io_context& ctx, const protocol_type& protocol);
```

2 *Effects:* Initializes the base class with `basic_socket<Protocol>(ctx, protocol)`.

```
basic_stream_socket(io_context& ctx, const endpoint_type& endpoint);
```

3 *Effects:* Initializes the base class with `basic_socket<Protocol>(ctx, endpoint)`.

```
basic_stream_socket(io_context& ctx, const protocol_type& protocol,
                    const native_handle_type& native_socket);
```

4 *Effects:* Initializes the base class with `basic_socket<Protocol>(ctx, protocol, native_socket)`.

```
basic_stream_socket(basic_stream_socket&& rhs);
```

5 *Effects:* Move constructs an object of class `basic_stream_socket<Protocol>`, initializing the base class with `basic_socket<Protocol>(std::move(rhs))`.

```
template<class OtherProtocol>
  basic_stream_socket(basic_stream_socket<OtherProtocol>&& rhs);
```

6 *Requires:* `OtherProtocol` is implicitly convertible to `Protocol`.

7 *Effects:* Move constructs an object of class `basic_stream_socket<Protocol>`, initializing the base class with `basic_socket<Protocol>(std::move(rhs))`.

8 *Remarks:* This constructor shall not participate in overload resolution unless `OtherProtocol` is implicitly convertible to `Protocol`.

18.8.2 basic_stream_socket assignment [socket.stream.assign]

```
basic_stream_socket& operator=(basic_stream_socket&& rhs);
```

1 *Effects:* Equivalent to `basic_socket<Protocol>::operator=(std::move(rhs))`.

2 *Returns:* `*this`.

```
template<class OtherProtocol>
  basic_stream_socket& operator=(basic_stream_socket<OtherProtocol>&& rhs);
```

3 *Requires:* `OtherProtocol` is implicitly convertible to `Protocol`.

4 *Effects:* Equivalent to `basic_socket<Protocol>::operator=(std::move(rhs))`.

5 *Returns:* `*this`.

6 *Remarks:* This assignment operator shall not participate in overload resolution unless `OtherProtocol` is implicitly convertible to `Protocol`.

18.8.3 basic_stream_socket operations [socket.stream.ops]

```
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 error_code& ec);
```

1 *Returns:* `receive(buffers, socket_base::message_flags(), ec)`.

```
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags);
template<class MutableBufferSequence>
  size_t receive(const MutableBufferSequence& buffers,
                 socket_base::message_flags flags, error_code& ec);
```

2 A read operation (16.2.4).

3 *Effects:* If `buffer_size(buffers) == 0`, returns immediately with no error. Otherwise, constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, and reads data from this socket as if by POSIX:

```
msghdr message;
message.msg_name = nullptr;
message.msg_namelen = 0;
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
recvmsg(native_handle(), &message, static_cast<int>(flags));
```

4 *Returns:* On success, the number of bytes received. Otherwise `0`.

5 *Error conditions:*

(5.1) — `stream_errc::eof` — if there is no data to be received and the peer performed an orderly shutdown.

```
template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive(const MutableBufferSequence& buffers,
                        CompletionToken&& token);
```

6 *Returns:* `async_receive(buffers, socket_base::message_flags(), forward<CompletionToken>(token))`.

```
template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_receive(const MutableBufferSequence& buffers,
                        socket_base::message_flags flags,
                        CompletionToken&& token);
```

7 A read operation (16.2.4).

8 *Completion signature:* `void(error_code ec, size_t n)`.

9 *Effects:* Initiates an asynchronous operation to read data from this socket. If `buffer_size(buffers) == 0`, the asynchronous operation completes immediately with no error and `n == 0`. Otherwise, constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, then reads data as if by POSIX:

```
msghdr message;
message.msg_name = nullptr;
message.msg_namelen = 0;
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
recvmsg(native_handle(), &message, static_cast<int>(flags));
```

10 If the operation completes successfully, `n` is the number of bytes received. Otherwise `n` is `0`.

11 *Error conditions:*

(11.1) — `errc::invalid_argument` — if `socket_base::message_peek` is set in flags.

(11.2) — `stream_errc::eof` — if there is no data to be received and the peer performed an orderly shutdown.

```
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers, error_code& ec);
```

12 *Returns:* `send(buffers, socket_base::message_flags(), ec)`.

```
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags);
template<class ConstBufferSequence>
  size_t send(const ConstBufferSequence& buffers,
              socket_base::message_flags flags, error_code& ec);
```

13 A write operation (16.2.4).

14 *Effects:* If `buffer_size(buffers) == 0`, returns immediately with no error. Otherwise, constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, and writes data to this socket as if by POSIX:

```
msghdr message;
message.msg_name = nullptr;
message.msg_namelen = 0;
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
sendmsg(native_handle(), &message, static_cast<int>(flags));
```

15 *Returns:* On success, the number of bytes sent. Otherwise `0`.

```
template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_send(const ConstBufferSequence& buffers, CompletionToken&& token);
```

16 *Returns:* `async_send(buffers, socket_base::message_flags(), forward<CompletionToken>(token))`.

```
template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_send(const ConstBufferSequence& buffers,
                     socket_base::message_flags flags,
                     CompletionToken&& token);
```

17 A write operation (16.2.4).

18 *Completion signature:* `void(error_code ec, size_t n)`.

19 *Effects:* Initiates an asynchronous operation to write data to this socket. If `buffer_size(buffers) == 0`, the asynchronous operation completes immediately with no error and `n == 0`. Otherwise, constructs an array `iov` of POSIX type `struct iovec` and length `iovlen`, corresponding to `buffers`, then writes data as if by POSIX:

```
msghdr message;
message.msg_name = nullptr;
message.msg_namelen = 0;
message.msg_iov = iov;
message.msg_iovlen = iovlen;
message.msg_control = nullptr;
message.msg_controllen = 0;
message.msg_flags = 0;
sendmsg(native_handle(), &message, static_cast<int>(flags));
```

20 If the operation completes successfully, `n` is the number of bytes sent. Otherwise `n` is `0`.

```
template<class MutableBufferSequence>
  size_t read_some(const MutableBufferSequence& buffers);
template<class MutableBufferSequence>
  size_t read_some(const MutableBufferSequence& buffers,
                   error_code& ec);
```

21 *Returns:* `receive(buffers, ec)`.

```
template<class MutableBufferSequence, class CompletionToken>
  DEDUCED async_read_some(const MutableBufferSequence& buffers,
                          CompletionToken&& token);
```

22 *Returns:* `async_receive(buffers, forward<CompletionToken>(token))`.

```
template<class ConstBufferSequence>
  size_t write_some(const ConstBufferSequence& buffers);
template<class ConstBufferSequence>
  size_t write_some(const ConstBufferSequence& buffers,
                    error_code& ec);
```

23 *Returns:* `send(buffers, ec)`.

```
template<class ConstBufferSequence, class CompletionToken>
  DEDUCED async_write_some(const ConstBufferSequence& buffers,
                           CompletionToken&& token);
```

24 *Returns:* `async_send(buffers, forward<CompletionToken>(token))`.

18.9 Class template `basic_socket_acceptor` [socket.acceptor]

1 An object of class template `basic_socket_acceptor<AcceptableProtocol>` is used to listen for, and queue, incoming socket connections. Socket objects that represent the incoming connections are dequeued by calling `accept` or `async_accept`.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class AcceptableProtocol>
  class basic_socket_acceptor : public socket_base
  {
  public:
    // types:

    using executor_type = io_context::executor_type;
    using native_handle_type = implementation defined; // see 18.2.3
    using protocol_type = AcceptableProtocol;
    using endpoint_type = typename protocol_type::endpoint;
    using socket_type = typename protocol_type::socket;

    // construct / copy / destroy:

    explicit basic_socket_acceptor(io_context& ctx);
    basic_socket_acceptor(io_context& ctx, const protocol_type& protocol);
    basic_socket_acceptor(io_context& ctx, const endpoint_type& endpoint,
                          bool reuse_addr = true);
    basic_socket_acceptor(io_context& ctx, const protocol_type& protocol,
                          const native_handle_type& native_acceptor);
    basic_socket_acceptor(const basic_socket_acceptor&) = delete;
    basic_socket_acceptor(basic_socket_acceptor&& rhs);
    template<class OtherProtocol>
      basic_socket_acceptor(basic_socket_acceptor<OtherProtocol>&& rhs);

    ~basic_socket_acceptor();

    basic_socket_acceptor& operator=(const basic_socket_acceptor&) = delete;
    basic_socket_acceptor& operator=(basic_socket_acceptor&& rhs);
    template<class OtherProtocol>
```

```
  basic_socket_acceptor& operator=(basic_socket_acceptor<OtherProtocol>&& rhs);

// basic_socket_acceptor operations:

executor_type get_executor() noexcept;

native_handle_type native_handle(); // see 18.2.3

void open(const protocol_type& protocol = protocol_type());
void open(const protocol_type& protocol, error_code& ec);

void assign(const protocol_type& protocol,
            const native_handle_type& native_acceptor); // see 18.2.3
void assign(const protocol_type& protocol,
            const native_handle_type& native_acceptor,
            error_code& ec); // see 18.2.3

bool is_open() const;

void close();
void close(error_code& ec);

void cancel();
void cancel(error_code& ec);

template<class SettableSocketOption>
  void set_option(const SettableSocketOption& option);
template<class SettableSocketOption>
  void set_option(const SettableSocketOption& option, error_code& ec);

template<class GettableSocketOption>
  void get_option(GettableSocketOption& option) const;
template<class GettableSocketOption>
  void get_option(GettableSocketOption& option, error_code& ec) const;

template<class IoControlCommand>
  void io_control(IoControlCommand& command);
template<class IoControlCommand>
  void io_control(IoControlCommand& command, error_code& ec);

void non_blocking(bool mode);
void non_blocking(bool mode, error_code& ec);
bool non_blocking() const;

void native_non_blocking(bool mode);
void native_non_blocking(bool mode, error_code& ec);
bool native_non_blocking() const;

void bind(const endpoint_type& endpoint);
void bind(const endpoint_type& endpoint, error_code& ec);

void listen(int backlog = max_listen_connections);
void listen(int backlog, error_code& ec);

endpoint_type local_endpoint() const;
```

```
    endpoint_type local_endpoint(error_code& ec) const;

    void enable_connection_aborted(bool mode);
    bool enable_connection_aborted() const;

    socket_type accept();
    socket_type accept(error_code& ec);
    socket_type accept(io_context& ctx);
    socket_type accept(io_context& ctx, error_code& ec);

    template<class CompletionToken>
      DEDUCED async_accept(CompletionToken&& token);
    template<class CompletionToken>
      DEDUCED async_accept(io_context& ctx, CompletionToken&& token);

    socket_type accept(endpoint_type& endpoint);
    socket_type accept(endpoint_type& endpoint, error_code& ec);
    socket_type accept(io_context& ctx, endpoint_type& endpoint);
    socket_type accept(io_context& ctx, endpoint_type& endpoint,
                       error_code& ec);

    template<class CompletionToken>
      DEDUCED async_accept(endpoint_type& endpoint,
                           CompletionToken&& token);
    template<class CompletionToken>
      DEDUCED async_accept(io_context& ctx, endpoint_type& endpoint,
                           CompletionToken&& token);

    void wait(wait_type w);
    void wait(wait_type w, error_code& ec);

    template<class CompletionToken>
      DEDUCED async_wait(wait_type w, CompletionToken&& token);

  private:
    protocol_type protocol_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Instances of class template `basic_socket_acceptor` meet the requirements of `Destructible` (C++Std [destructible]), `MoveConstructible` (C++Std [moveconstructible]), and `MoveAssignable` (C++Std [moveassignable]).

3 When there are multiple outstanding asynchronous accept operations the order in which the incoming connections are dequeued, and the order of invocation of the completion handlers for these operations, is unspecified.

4 When an operation has its effects specified as if by passing the result of `native_handle()` to a POSIX function, then the operation fails with error condition `errc::bad_file_descriptor` if `is_open() == false` at the point in the effects when the POSIX function is called.

18.9.1 `basic_socket_acceptor` constructors [socket.acceptor.cons]

```
explicit basic_socket_acceptor(io_context& ctx);
```

1 *Postconditions:*

(1.1) — `get_executor() == ctx.get_executor()`.

(1.2) — `is_open() == false`.

```
basic_socket_acceptor(io_context& ctx, const protocol_type& protocol);
```

2 *Effects:* Opens this acceptor as if by calling `open(protocol)`.

3 *Postconditions:*

(3.1) — `get_executor() == ctx.get_executor()`.

(3.2) — `is_open() == true`.

(3.3) — `non_blocking() == false`.

(3.4) — `enable_connection_aborted() == false`.

(3.5) — `protocol_ == protocol`.

```
basic_socket_acceptor(io_context& ctx, const endpoint_type& endpoint,
                      bool reuse_addr = true);
```

4 *Effects:* Opens and binds this acceptor as if by calling:

```
open(endpoint.protocol());
if (reuse_addr)
  set_option(reuse_address(true));
bind(endpoint);
listen();
```

5 *Postconditions:*

(5.1) — `get_executor() == ctx.get_executor()`.

(5.2) — `is_open() == true`.

(5.3) — `non_blocking() == false`.

(5.4) — `enable_connection_aborted() == false`.

(5.5) — `protocol_ == endpoint.protocol()`.

```
basic_socket_acceptor(io_context& ctx, const protocol_type& protocol,
                      const native_handle_type& native_acceptor);
```

6 *Requires:* `native_acceptor` is a native handle to an open acceptor.

7 *Effects:* Assigns the existing native acceptor into this acceptor as if by calling `assign(protocol, native_acceptor)`.

8 *Postconditions:*

(8.1) — `get_executor() == ctx.get_executor()`.

(8.2) — `is_open() == true`.

(8.3) — `non_blocking() == false`.

(8.4) — `enable_connection_aborted() == false`.

(8.5) — `protocol_ == protocol`.

```
basic_socket_acceptor(basic_socket_acceptor&& rhs);
```

9 *Effects:* Move constructs an object of class `basic_socket_acceptor<AcceptableProtocol>` that refers to the state originally represented by `rhs`.

10 *Postconditions:*

(10.1) — `get_executor() == rhs.get_executor()`.

(10.2) — `is_open()` returns the same value as `rhs.is_open()` prior to the constructor invocation.

(10.3) — `non_blocking()` returns the same value as `rhs.non_blocking()` prior to the constructor invocation.

(10.4) — `enable_connection_aborted()` returns the same value as `rhs.enable_connection_aborted()` prior to the constructor invocation.

(10.5) — `protocol_` is equal to the prior value of `rhs.protocol_`.

(10.6) — `rhs.is_open() == false`.

```
template<class OtherProtocol>
  basic_socket_acceptor(basic_socket_acceptor<OtherProtocol>&& rhs);
```

11 *Requires:* `OtherProtocol` is implicitly convertible to `Protocol`.

12 *Effects:* Move constructs an object of class `basic_socket_acceptor<AcceptableProtocol>` that refers to the state originally represented by `rhs`.

13 *Postconditions:*

(13.1) — `get_executor() == rhs.get_executor()`.

(13.2) — `is_open()` returns the same value as `rhs.is_open()` prior to the constructor invocation.

(13.3) — `non_blocking()` returns the same value as `rhs.non_blocking()` prior to the constructor invocation.

(13.4) — `enable_connection_aborted()` returns the same value as `rhs.enable_connection_aborted()` prior to the constructor invocation.

(13.5) — `native_handle()` returns the prior value of `rhs.native_handle()`.

(13.6) — `protocol_` is the result of converting the prior value of `rhs.protocol_`.

(13.7) — `rhs.is_open() == false`.

14 *Remarks:* This constructor shall not participate in overload resolution unless `OtherProtocol` is implicitly convertible to `Protocol`.

18.9.2 `basic_socket_acceptor` destructor [socket.acceptor.dtor]

```
~basic_socket_acceptor();
```

1 *Effects:* If `is_open()` is `true`, cancels all outstanding asynchronous operations associated with this acceptor, and releases acceptor resources as if by POSIX `close(native_handle())`. Completion handlers for canceled operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

18.9.3 `basic_socket_acceptor` assignment [socket.acceptor.assign]

```
basic_socket_acceptor& operator=(basic_socket_acceptor&& rhs);
```

1 *Effects:* If `is_open()` is `true`, cancels all outstanding asynchronous operations associated with this acceptor, and releases acceptor resources as if by POSIX `close(native_handle())`. Then moves into `*this` the state originally represented by `rhs`. Completion handlers for canceled operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

2 *Postconditions:*

(2.1) — `get_executor() == rhs.get_executor()`.

(2.2) — `is_open()` returns the same value as `rhs.is_open()` prior to the assignment.

(2.3) — `non_blocking()` returns the same value as `rhs.non_blocking()` prior to the assignment.

(2.4) — `enable_connection_aborted()` returns the same value as `rhs.enable_connection_aborted()` prior to the assignment.

(2.5) — `native_handle()` returns the same value as `rhs.native_handle()` prior to the assignment.

(2.6) — `protocol_` is the same value as `rhs.protocol_` prior to the assignment.

(2.7) — `rhs.is_open() == false`.

3 *Returns:* `*this`.

```
template<class OtherProtocol>
  basic_socket_acceptor& operator=(basic_socket_acceptor<OtherProtocol>&& rhs);
```

4 *Requires:* `OtherProtocol` is implicitly convertible to `Protocol`.

5 *Effects:* If `is_open()` is `true`, cancels all outstanding asynchronous operations associated with this acceptor, and releases acceptor resources as if by POSIX `close(native_handle())`. Then moves into `*this` the state originally represented by `rhs`. Completion handlers for canceled operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

6 *Postconditions:*

(6.1) — `get_executor() == rhs.get_executor()`.

(6.2) — `is_open()` returns the same value as `rhs.is_open()` prior to the assignment.

(6.3) — `non_blocking()` returns the same value as `rhs.non_blocking()` prior to the assignment.

(6.4) — `enable_connection_aborted()` returns the same value as `rhs.enable_connection_aborted()` prior to the assignment.

(6.5) — `native_handle()` returns the same value as `rhs.native_handle()` prior to the assignment.

(6.6) — `protocol_` is the result of converting the value of `rhs.protocol_` prior to the assignment.

(6.7) — `rhs.is_open() == false`.

7 *Returns:* `*this`.

8 *Remarks:* This assignment operator shall not participate in overload resolution unless `OtherProtocol` is implicitly convertible to `Protocol`.

18.9.4 `basic_socket_acceptor` operations [socket.acceptor.ops]

```
executor_type get_executor() noexcept;
```

1 *Returns:* The associated executor.

```
native_handle_type native_handle();
```

2 *Returns:* The native representation of this acceptor.

```
void open(const protocol_type& protocol);
void open(const protocol_type& protocol, error_code& ec);
```

3 *Effects:* Establishes the postcondition, as if by POSIX `socket(protocol.family(), protocol.type(), protocol.protocol())`.

4 *Postconditions:*

(4.1) — `is_open() == true`.

(4.2) — `non_blocking() == false`.

(4.3) — `enable_connection_aborted() == false`.

(4.4) — `protocol_ == protocol`.

5 *Error conditions:*

(5.1) — `socket_errc::already_open` — if `is_open()` is `true`.

```
void assign(const protocol_type& protocol,
            const native_handle_type& native_acceptor);
void assign(const protocol_type& protocol,
            const native_handle_type& native_acceptor, error_code& ec);
```

6 *Requires:* `native_acceptor` is a native handle to an open acceptor.

7 *Effects:* Assigns the native acceptor handle to this acceptor object.

8 *Postconditions:*

(8.1) — `is_open() == true`.

(8.2) — `non_blocking() == false`.

(8.3) — `enable_connection_aborted() == false`.

(8.4) — `protocol_ == protocol`.

9 *Error conditions:*

(9.1) — `socket_errc::already_open` — if `is_open()` is `true`.

```
bool is_open() const;
```

10 *Returns:* A `bool` indicating whether this acceptor was opened by a previous call to `open` or `assign`.

```
void close();
void close(error_code& ec);
```

11 *Effects:* If `is_open()` is `true`, cancels all outstanding asynchronous operations associated with this acceptor, and establishes the postcondition as if by POSIX `close(native_handle())`. Completion handlers for canceled asynchronous operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

12 *Postconditions:* `is_open() == false`.

```
void cancel();
void cancel(error_code& ec);
```

13 *Effects:* Cancels all outstanding asynchronous operations associated with this acceptor. Completion handlers for canceled asynchronous operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

14 *Error conditions:*

(14.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

(14.2) — `errc::operation_not_supported` — current conditions do not permit cancelation. The conditions under which cancelation of asynchronous operations is permitted are implementation-defined.

```
template<class SettableSocketOption>
  void set_option(const SettableSocketOption& option);
template<class SettableSocketOption>
  void set_option(const SettableSocketOption& option, error_code& ec);
```

15 *Effects:* Sets an option on this acceptor, as if by POSIX `setsockopt(native_handle(), option.level(protocol_), option.name(protocol_), option.data(protocol_), option.size(protocol_))`.

```
template<class GettableSocketOption>
  void get_option(GettableSocketOption& option);
template<class GettableSocketOption>
  void get_option(GettableSocketOption& option, error_code& ec);
```

16 *Effects:* Gets an option from this acceptor, as if by POSIX:

```
socklen_t option_len = option.size(protocol_);
int result = getsockopt(native_handle(), option.level(protocol_),
                        option.name(protocol_), option.data(protocol_),
                        &option_len);
if (result == 0)
  option.resize(option_len);
```

```
template<class IoControlCommand>
  void io_control(IoControlCommand& command);
template<class IoControlCommand>
  void io_control(IoControlCommand& command, error_code& ec);
```

17 *Effects:* Executes an I/O control command on this acceptor, as if by POSIX `ioctl(native_handle(), command.name(), command.data())`.

```
void non_blocking(bool mode);
void non_blocking(bool mode, error_code& ec);
```

18 *Effects:* Sets the non-blocking mode of this acceptor. The non-blocking mode determines whether subsequent synchronous socket operations (18.2.1) on `*this` block the calling thread.

19 *Error conditions:*

(19.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

20 *Postconditions:* `non_blocking() == mode`.

21 [*Note:* The non-blocking mode has no effect on the behavior of asynchronous operations. — *end note*]

```
bool non_blocking() const;
```

22 *Returns:* The non-blocking mode of this acceptor.

```
void native_non_blocking(bool mode);
void native_non_blocking(bool mode, error_code& ec);
```

23 *Effects:* Sets the non-blocking mode of the underlying native acceptor, as if by POSIX:

```
int flags = fcntl(native_handle(), F_GETFL, 0);
if (flags >= 0)
{
  if (mode)
    flags |= O_NONBLOCK;
  else
    flags &= ~O_NONBLOCK;
  fcntl(native_handle(), F_SETFL, flags);
}
```

24 The native non-blocking mode has no effect on the behavior of the synchronous or asynchronous operations specified in this clause.

25 *Error conditions:*

(25.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

(25.2) — `errc::invalid_argument` — if `mode == false` and `non_blocking() == true`. [*Note:* As the combination does not make sense. — *end note*]

```
bool native_non_blocking() const;
```

26 *Returns:* The non-blocking mode of the underlying native acceptor.

27 *Remarks:* Implementations are permitted and encouraged to cache the native non-blocking mode that was applied through a prior call to `native_non_blocking`. Implementations may return an incorrect value if a program sets the non-blocking mode directly on the acceptor, by calling an operating system-specific function on the result of `native_handle()`.

```
void bind(const endpoint_type& endpoint);
void bind(const endpoint_type& endpoint, error_code& ec);
```

28 *Effects:* Binds this acceptor to the specified local endpoint, as if by POSIX `bind(native_handle(), endpoint.data(), endpoint.size())`.

```
void listen(int backlog = socket_base::max_listen_connections);
void listen(int backlog, error_code& ec);
```

29 *Effects:* Marks this acceptor as ready to accept connections, as if by POSIX `listen(native_handle(), backlog)`.

```
endpoint_type local_endpoint() const;
endpoint_type local_endpoint(error_code& ec) const;
```

30 *Effects:* Determines the locally-bound endpoint associated with this acceptor, as if by POSIX:

```
endpoint_type endpoint;
socklen_t endpoint_len = endpoint.capacity();
int result = getsockname(native_handle(), endpoint.data(), &endpoint_len);
if (result == 0)
  endpoint.resize(endpoint_len);
```

31 *Returns:* On success, `endpoint`. Otherwise `endpoint_type()`.

```
void enable_connection_aborted(bool mode);
```

32 *Effects:* If `mode` is true, subsequent synchronous or asynchronous accept operations on this acceptor are permitted to fail with error condition `errc::connection_aborted`. If `mode` is `false`, subsequent accept operations will not fail with `errc::connection_aborted`. [*Note:* If `mode` is `false`, the implementation will restart the call to POSIX `accept` if it fails with `ECONNABORTED`. — *end note*]

33 *Error conditions:*

(33.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

```
bool enable_connection_aborted() const;
```

34 *Returns:* Whether accept operations on this acceptor are permitted to fail with `errc::connection_aborted`.

```
socket_type accept();
socket_type accept(error_code& ec);
```

35 *Returns:* `accept(get_executor().context(), ec)`.

```
socket_type accept(io_context& ctx);
socket_type accept(io_context& ctx, error_code& ec);
```

36 *Effects:* Extracts a socket from the queue of pending connections of the acceptor, as if by POSIX:

```
native_handle_type h = accept(native_handle(), nullptr, 0);
```

37 *Returns:* On success, `socket_type(ctx, protocol_, h)`. Otherwise `socket_type(ctx)`.

```
template<class CompletionToken>
  DEDUCED async_accept(CompletionToken&& token);
```

38 *Returns:* `async_accept(get_executor().context(), forward<CompletionToken>(token))`.

```
template<class CompletionToken>
  DEDUCED async_accept(io_context& ctx, CompletionToken&& token);
```

39 *Completion signature:* `void(error_code ec, socket_type s)`.

40 *Effects:* Initiates an asynchronous operation to extract a socket from the queue of pending connections of the acceptor, as if by POSIX:

```
native_handle_type h = accept(native_handle(), nullptr, 0);
```

On success, `s` is `socket_type(ctx, protocol_, h)`. Otherwise, `s` is `socket_type(ctx)`.

```
socket_type accept(endpoint_type& endpoint);
socket_type accept(endpoint_type& endpoint, error_code& ec);
```

41 *Returns:* `accept(get_executor().context(), endpoint, ec)`.

```
socket_type accept(io_context& ctx, endpoint_type& endpoint);
socket_type accept(io_context& ctx, endpoint_type& endpoint,
                   error_code& ec);
```

42 *Effects:* Extracts a socket from the queue of pending connections of the acceptor, as if by POSIX:

```
socklen_t endpoint_len = endpoint.capacity();
native_handle_type h = accept(native_handle(),
                              endpoint.data(),
                              &endpoint_len);
if (h >= 0)
  endpoint.resize(endpoint_len);
```

43 *Returns:* On success, `socket_type(ctx, protocol_, h)`. Otherwise `socket_type(ctx)`.

```
template<class CompletionToken>
  DEDUCED async_accept(endpoint_type& endpoint,
                       CompletionToken&& token);
```

44 *Returns:* `async_accept(get_executor().context(), endpoint, forward<CompletionToken>(token))`.

```
template<class CompletionToken>
  DEDUCED async_accept(io_context& ctx, endpoint_type& endpoint,
                       CompletionToken&& token);
```

45 *Completion signature:* `void(error_code ec, socket_type s)`.

46 *Effects:* Initiates an asynchronous operation to extract a socket from the queue of pending connections of the acceptor, as if by POSIX:

```
socklen_t endpoint_len = endpoint.capacity();
native_handle_type h = accept(native_handle(),
                              endpoint.data(),
                              &endpoint_len);
if (h >= 0)
  endpoint.resize(endpoint_len);
```

On success, `s` is `socket_type(ctx, protocol_, h)`. Otherwise, `s` is `socket_type(ctx)`.

```
void wait(wait_type w);
void wait(wait_type w, error_code& ec);
```

47 *Effects:* Waits for the acceptor to have a queued incoming connection, or to have error conditions pending, as if by POSIX `poll`.

```
template<class CompletionToken>
  DEDUCED async_wait(wait_type w, CompletionToken&& token);
```

48 *Completion signature:* `void(error_code ec)`.

49 *Effects:* Initiates an asynchronous operation to wait for the acceptor to have a queued incoming connection, or to have error conditions pending, as if by POSIX `poll`.

50 When multiple asynchronous wait operations are initiated with the same `wait_type` value, all outstanding operations complete when the acceptor enters the corresponding ready state. The order of invocation of the completions handlers for these operations is unspecified.

51 *Error conditions:*

(51.1) — `errc::bad_file_descriptor` — if `is_open()` is `false`.

19 Socket iostreams [socket.iostreams]

19.1 Class template `basic_socket_streambuf` [socket.streambuf]

1 The class `basic_socket_streambuf<Protocol, Clock, WaitTraits>` associates both the input sequence and the output sequence with a socket. The input and output sequences do not support seeking. [*Note:* The input and output sequences are independent as a stream socket provides full duplex I/O. — *end note*]

2 [*Note:* This class is intended for sending and receiving bytes, not characters. The conversion from characters to bytes, and vice versa, must occur elsewhere. — *end note*]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Protocol, class Clock, class WaitTraits>
  class basic_socket_streambuf : public basic_streambuf<char>
  {
  public:
    // types:

    using protocol_type = Protocol;
    using endpoint_type = typename protocol_type::endpoint;
    using clock_type = Clock;
    using time_point = typename clock_type::time_point;
    using duration = typename clock_type::duration;
    using wait_traits_type = WaitTraits;

    // construct / copy / destroy:

    basic_socket_streambuf();
    explicit basic_socket_streambuf(basic_stream_socket<protocol_type> s);
    basic_socket_streambuf(const basic_socket_streambuf&) = delete;
    basic_socket_streambuf(basic_socket_streambuf&& rhs);

    virtual ~basic_socket_streambuf();

    basic_socket_streambuf& operator=(const basic_socket_streambuf&) = delete;
    basic_socket_streambuf& operator=(basic_socket_streambuf&& rhs);

    // members:

    basic_socket_streambuf* connect(const endpoint_type& e);
    template<class... Args> basic_socket_streambuf* connect(Args&&... );

    basic_socket_streambuf* close();

    basic_socket<protocol_type>& socket();
    error_code error() const;

    time_point expiry() const;
    void expires_at(const time_point& t);
```

```
    void expires_after(const duration& d);

  protected:
    // overridden virtual functions:
    virtual int_type underflow() override;
    virtual int_type pbackfail(int_type c = traits_type::eof()) override;
    virtual int_type overflow(int_type c = traits_type::eof()) override;
    virtual int sync() override;
    virtual streambuf* setbuf(char_type* s, streamsize n) override;

  private:
    basic_stream_socket<protocol_type> socket_; // exposition only
    error_code ec_; // exposition only
    time_point expiry_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

3 Instances of class template `basic_socket_streambuf` meet the requirements of `Destructible` (C++Std [destructible]), `MoveConstructible` (C++Std [moveconstructible]), and `MoveAssignable` (C++Std [moveassignable]).

19.1.1 `basic_socket_streambuf` constructors [socket.streambuf.cons]

`basic_socket_streambuf();`

1 *Effects:* Initializes `socket_` with `ctx`, where `ctx` is an unspecified object of class `io_context`.

2 *Postconditions:* `expiry() == time_point::max()`.

`explicit basic_socket_streambuf(basic_stream_socket<protocol_type> s);`

3 *Effects:* Initializes `socket_` with `std::move(s)`.

4 *Postconditions:* `expiry() == time_point::max()`.

`basic_socket_streambuf(basic_socket_streambuf&& rhs);`

5 *Effects:* Move constructs from the rvalue `rhs`. It is implementation-defined whether the sequence pointers in `*this` (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) obtain the values which `rhs` had. Whether they do or not, `*this` and `rhs` reference separate buffers (if any at all) after the construction. Additionally `*this` references the socket which `rhs` did before the construction, and `rhs` references no open socket after the construction.

6 *Postconditions:* Let `rhs_p` refer to the state of `rhs` just prior to this construction and let `rhs_a` refer to the state of `rhs` just after this construction.

(6.1) — `is_open() == rhs_p.is_open()`

(6.2) — `rhs_a.is_open() == false`

(6.3) — `expiry() == rhs_p.expiry()`

(6.4) — `rhs_a.expiry() == time_point::max()`

(6.5) — `gptr() - eback() == rhs_p.gptr() - rhs_p.eback()`

(6.6) — `egptr() - eback() == rhs_p.egptr() - rhs_p.eback()`

(6.7) — `ptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()`

(6.8) — `pptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()`

(6.9) — `if (eback()) eback() != rhs_a.eback()`

(6.10) — `if (gptr()) gptr() != rhs_a.gptr()`

(6.11) — `if (egptr()) egptr() != rhs_a.egptr()`

(6.12) — `if (pbase()) pbase() != rhs_a.pbase()`

(6.13) — `if (pptr()) pptr() != rhs_a.pptr()`

(6.14) — `if (epptr()) epptr() != rhs_a.epptr()`

```
virtual ~basic_socket_streambuf();
```

7 *Effects:* If a put area exists, calls `overflow(traits_type::eof())` to flush characters. [*Note:* The socket is closed by the `basic_stream_socket<protocol_type>` destructor. — *end note*]

```
basic_socket_streambuf& operator=(basic_socket_streambuf&& rhs);
```

8 *Effects:* Calls `this->close()` then move assigns from `rhs`. After the move assignment `*this` has the observable state it would have had if it had been move constructed from `rhs`.

9 *Returns:* `*this`.

19.1.2 `basic_socket_streambuf` members [socket.streambuf.members]

```
basic_socket_streambuf* connect(const endpoint_type& e);
```

1 *Effects:* Initializes the `basic_socket_streambuf` as required, closes and re-opens the socket by performing `socket_.close(ec_)` and `socket_.open(e.protocol(), ec_)`, then attempts to establish a connection as if by POSIX `connect(socket_.native_handle(), static_cast<sockaddr*>(e.data()), e.size())`. `ec_` is set to reflect the error code produced by the operation. If the operation does not complete before the absolute timeout specified by `expiry_`, the socket is closed and `ec_` is set to `errc::timed_out`.

2 *Returns:* if `!ec_`, `this`; otherwise, a null pointer.

```
template<class... Args>
  basic_socket_streambuf* connect(Args&&... args);
```

3 *Effects:* Initializes the `basic_socket_streambuf` as required and closes the socket as if by calling `socket_.close(ec_)`. Obtains an endpoint sequence `endpoints` by performing `protocol_type::resolver(ctx).resolve(forward<Args>(args)...)`, where `ctx` is an unspecified object of class `io_context`. For each endpoint `e` in the sequence, closes and re-opens the socket by performing `socket_.close(ec_)` and `socket_.open(e.protocol(), ec_)`, then attempts to establish a connection as if by POSIX `connect(socket_.native_handle(), static_cast<sockaddr*>(e.data()), e.size())`. `ec_` is set to reflect the error code produced by the operation. If the operation does not complete before the absolute timeout specified by `expiry_`, the socket is closed and `ec_` is set to `errc::timed_out`.

4 *Returns:* if `!ec_`, `this`; otherwise, a null pointer.

5 *Remarks:* This function shall not participate in overload resolution unless `Protocol` meets the requirements for an internet protocol (21.2.1).

```
basic_socket_streambuf* close();
```

6 *Effects:* If a put area exists, calls `overflow(traits_type::eof())` to flush characters. Regardless of whether the preceding call fails or throws an exception, the function closes the socket as if by `basic_socket<protocol_type>::close(ec_)`. If any of the calls made by the function fail, `close` fails by returning a null pointer. If one of these calls throws an exception, the exception is caught and rethrown after closing the socket.

7 *Returns:* `this` on success, a null pointer otherwise.

8 *Postconditions:* `is_open() == false`.

```
basic_socket<protocol_type>& socket();
```

9 *Returns:* `socket_`.

```
error_code error() const;
```

10 *Returns:* `ec_`.

```
time_point expiry() const;
```

11 *Returns:* `expiry_`.

```
void expires_at(const time_point& t);
```

12 *Postconditions:* `expiry_ == t`.

```
void expires_after(const duration& d);
```

13 *Effects:* Equivalent to `expires_at(clock_type::now() + d)`.

19.1.3 `basic_socket_streambuf` overridden virtual functions [socket.streambuf.virtual]

```
virtual int_type underflow() override;
```

1 *Effects:* Behaves according to the description of `basic_streambuf<char>::underflow()`, with the specialization that a sequence of characters is read from the input sequence as if by POSIX `recvmsg`, and `ec_` is set to reflect the error code produced by the operation. If the operation does not complete before the absolute timeout specified by `expiry_`, the socket is closed and `ec_` is set to `errc::timed_out`.

2 *Effects:* Returns `traits_type::eof()` to indicate failure. Otherwise returns `traits_type::to_int_type(*gptr())`.

```
virtual int_type pbackfail(int_type c = traits_type::eof()) override;
```

3 *Effects:* Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

(3.1) — If `traits_type::eq_int_type(c,traits_type::eof())` returns `false`, and if the function makes a putback position available, and if `traits_type::eq(traits_type::to_char_type(c),gptr()[-1])` returns `true`, decrements the next pointer for the input sequence, `gptr()`. Returns: `c`.

(3.2) — If `traits_type::eq_int_type(c,traits_type::eof())` returns `false`, and if the function makes a putback position available, and if the function is permitted to assign to the putback position, decrements the next pointer for the input sequence, and stores `c` there. Returns: `c`.

(3.3) — If `traits_type::eq_int_type(c,traits_type::eof())` returns `true`, and if either the input sequence has a putback position available or the function makes a putback position available, decrements the next pointer for the input sequence, `gptr()`. Returns: `traits_type::not_eof(c)`.

4 *Returns:* `traits_type::eof()` to indicate failure.

5 Notes: The function does not put back a character directly to the input sequence. If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

```
virtual int_type overflow(int_type c = traits_type::eof()) override;
```

6 *Effects:* Behaves according to the description of `basic_streambuf<char>::overflow(c)`, except that the behavior of "consuming characters" is performed by output of the characters to the socket as if by one or more calls to POSIX `sendmsg`, and `ec_` is set to reflect the error code produced by the operation. If the operation does not complete before the absolute timeout specified by `expiry_`, the socket is closed and `ec_` is set to `errc::timed_out`.

7 *Returns:* `traits_type::not_eof(c)` to indicate success, and `traits_type::eof()` to indicate failure.

```
virtual int sync() override;
```

8 *Effects:* If a put area exists, calls `overflow(traits_type::eof())` to flush characters.

```
virtual streambuf* setbuf(char_type* s, streamsize n) override;
```

9 *Effects:* If `setbuf(nullptr, 0)` is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are unspecified. "Unbuffered" means that `pbase()` and `pptr()` always return null and output to the socket should appear as soon as possible.

19.2 Class template `basic_socket_iostream` [socket.iostream]

1 The class template `basic_socket_iostream<Protocol, Clock, WaitTraits>` supports reading and writing on sockets. It uses a `basic_socket_streambuf<Protocol, Clock, WaitTraits>` object to control the associated sequences.

2 [*Note:* This class is intended for sending and receiving bytes, not characters. The conversion from characters to bytes, and vice versa, must occur elsewhere. — *end note*]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {

  template<class Protocol, class Clock, class WaitTraits>
  class basic_socket_iostream : public basic_iostream<char>
  {
  public:
    // types:

    using protocol_type = Protocol;
    using endpoint_type = typename protocol_type::endpoint;
    using clock_type = Clock;
    using time_point = typename clock_type::time_point;
    using duration = typename clock_type::duration;
    using wait_traits_type = WaitTraits;

    // construct / copy / destroy:

    basic_socket_iostream();
    explicit basic_socket_iostream(basic_stream_socket<protocol_type> s);
    basic_socket_iostream(const basic_socket_iostream&) = delete;
    basic_socket_iostream(basic_socket_iostream&& rhs);
    template<class... Args>
      explicit basic_socket_iostream(Args&&... args);

    basic_socket_iostream& operator=(const basic_socket_iostream&) = delete;
    basic_socket_iostream& operator=(basic_socket_iostream&& rhs);
```

```
    // members:

    template<class... Args> void connect(Args&&... args);

    void close();

    basic_socket_streambuf<protocol_type, clock_type, wait_traits_type>* rdbuf() const;

    basic_socket<protocol_type>& socket();
    error_code error() const;

    time_point expiry() const;
    void expires_at(const time_point& t);
    void expires_after(const duration& d);

  private:
    basic_socket_streambuf<protocol_type, clock_type, wait_traits_type> sb_; // exposition only
  };

} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

3 Instances of class template `basic_socket_iostream` meet the requirements of `Destructible` (C++Std [destructible]), `MoveConstructible` (C++Std [moveconstructible]), and `MoveAssignable` (C++Std [moveassignable]).

19.2.1 `basic_socket_iostream` constructors [socket.iostream.cons]

`basic_socket_iostream();`

1 *Effects:* Initializes the base class as `basic_iostream<char>(&sb_)`, `sb_` as `basic_socket_streambuf<Protocol, Clock, WaitTraits>()`, and performs `setf(std::ios_base::unitbuf)`.

`explicit basic_socket_iostream(basic_stream_socket<protocol_type> s);`

2 *Effects:* Initializes the base class as `basic_iostream<char>(&sb_)`, `sb_` as `basic_socket_streambuf<Protocol, Clock, WaitTraits>(std::move(s))`, and performs `setf(std::ios_base::unitbuf)`.

`basic_socket_iostream(basic_socket_iostream&& rhs);`

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_socket_streambuf`. Next `basic_iostream<char>::set_rdbuf(&sb_)` is called to install the contained `basic_socket_streambuf`.

```
template<class... Args>
  explicit basic_socket_iostream(Args&&... args);
```

4 *Effects:* Initializes the base class as `basic_iostream<char>(&sb_)`, initializes `sb_` as `basic_socket_streambuf<Protocol, Clock, WaitTraits>()`, and performs `setf(std::ios_base::unitbuf)`. Then calls `rdbuf()->connect(forward<Args>(args)...)`. If that function returns a null pointer, calls `setstate(failbit)`.

`basic_socket_iostream& operator=(basic_socket_iostream&& rhs);`

5 *Effects:* Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

6 *Returns:* `*this`.

19.2.2 `basic_socket_iostream` members [socket.iostream.members]

```
template<class... Args>
  void connect(Args&&... args);
```

1 *Effects:* Calls `rdbuf()->connect(forward<Args>(args)...)`. If that function returns a null pointer, calls `setstate(failbit)` (which may throw `ios_base::failure`).

```
void close();
```

2 *Effects:* Calls `rdbuf()->close()`. If that function returns a null pointer, calls `setstate(failbit)` (which may throw `ios_base::failure`).

```
basic_socket_streambuf<protocol_type, clock_type, wait_traits_type>* rdbuf() const;
```

3 *Returns:* `const_cast<basic_socket_streambuf<protocol_type, clock_type, wait_traits_type>*>(std::addre` `))`.

```
basic_socket<protocol_type>& socket();
```

4 *Returns:* `rdbuf()->socket()`.

```
error_code error() const;
```

5 *Returns:* `rdbuf()->error()`.

```
time_point expiry() const;
```

6 *Returns:* `rdbuf()->expiry()`.

```
void expires_at(const time_point& t);
```

7 *Effects:* Equivalent to `rdbuf()->expires_at(t)`.

```
void expires_after(const duration& d);
```

8 *Effects:* Equivalent to `rdbuf()->expires_after(d)`.

20 Socket algorithms [socket.algo]

20.1 Synchronous connect operations [socket.algo.connect]

```
template<class Protocol, class EndpointSequence>
  typename Protocol::endpoint connect(basic_socket<Protocol>& s,
                                      const EndpointSequence& endpoints);
template<class Protocol, class InputIterator>
  typename Protocol::endpoint connect(basic_socket<Protocol>& s,
                                      const EndpointSequence& endpoints,
                                      error_code& ec);
```

1 *Returns:* `connect(s, endpoints, [](auto, auto){ return true; }, ec)`.

```
template<class Protocol, class EndpointSequence, class ConnectCondition>
  typename Protocol::endpoint connect(basic_socket<Protocol>& s,
                                      const EndpointSequence& endpoints,
                                      ConnectCondition c);
template<class Protocol, class InputIterator, class ConnectCondition>
  typename Protocol::endpoint connect(basic_socket<Protocol>& s,
                                      const EndpointSequence& endpoints,
                                      ConnectCondition c, error_code& ec);
```

2 *Effects:* Performs `ec.clear()`, then finds the first element `ep` in the sequence `endpoints` for which:

(2.1) — `c(ec, ep)` yields `true`;

(2.2) — `s.close(ec)` succeeds;

(2.3) — `s.open(ep.protocol(), ec)` succeeds; and

(2.4) — `s.connect(ep, ec)` succeeds.

3 *Returns:* `typename Protocol::endpoint()` if no such element is found, otherwise `ep`.

4 *Error conditions:*

(4.1) — `socket_errc::not_found` — if `endpoints.empty()` or if the function object `c` returned `false` for all elements in the sequence.

```
template<class Protocol, class InputIterator>
  InputIterator connect(basic_socket<Protocol>& s,
                        InputIterator first, InputIterator last);
template<class Protocol, class InputIterator>
  InputIterator connect(basic_socket<Protocol>& s,
                        InputIterator first, InputIterator last,
                        error_code& ec);
```

5 *Returns:* `connect(s, first, last, [](auto, auto){ return true; }, ec)`.

```
template<class Protocol, class InputIterator, class ConnectCondition>
  InputIterator connect(basic_socket<Protocol>& s,
                        InputIterator first, InputIterator last,
                        ConnectCondition c);
template<class Protocol, class InputIterator, class ConnectCondition>
  InputIterator connect(basic_socket<Protocol>& s,
```

```
                    InputIterator first, InputIterator last,
                    ConnectCondition c, error_code& ec);
```

6 *Effects:* Performs `ec.clear()`, then finds the first iterator `i` in the range `[first, last)` for which:

(6.1) — `c(ec, *i)` yields `true`;

(6.2) — `s.close(ec)` succeeds;

(6.3) — `s.open(typename Protocol::endpoint(*i).protocol(), ec)` succeeds; and

(6.4) — `s.connect(*i, ec)` succeeds.

7 *Returns:* `last` if no such iterator is found, otherwise `i`.

8 *Error conditions:*

(8.1) — `socket_errc::not_found` — if `first == last` or if the function object `c` returned `false` for all iterators in the range.

20.2 Asynchronous connect operations [socket.algo.async.connect]

```
template<class Protocol, class EndpointSequence, class CompletionToken>
  DEDUCED async_connect(basic_socket<Protocol>& s,
                        const EndpointSequence& endpoints,
                        CompletionToken&& token);
```

1 *Returns:* `async_connect(s, endpoints, [](auto, auto){ return true; }, forward<CompletionToken>(token))`.

```
template<class Protocol, class InputIterator,
  class ConnectCondition, class CompletionToken>
    DEDUCED async_connect(basic_socket<Protocol>& s,
                          const EndpointSequence& endpoints,
                          ConnectCondition c,
                          CompletionToken&& token);
```

2 A composed asynchronous operation (13.2.7.14).

3 *Completion signature:* `void(error_code ec, typename Protocol::endpoint ep)`.

4 *Effects:* Performs `ec.clear()`, then finds the first element `ep` in the sequence `endpoints` for which:

(4.1) — `c(ec, ep)` yields `true`;

(4.2) — `s.close(ec)` succeeds;

(4.3) — `s.open(ep.protocol(), ec)` succeeds; and

(4.4) — the asynchronous operation `s.async_connect(ep, `*`unspecified`*`)` succeeds.

5 `ec` is updated with the result of the `s.async_connect(ep, `*`unspecified`*`)` operation, if any. If no such element is found, or if the operation fails with one of the error conditions listed below, `ep` is set to `typename Protocol::endpoint()`. [*Note:* The underlying `close`, `open`, and `async_connect` operations are performed sequentially. — *end note*]

6 *Error conditions:*

(6.1) — `socket_errc::not_found` — if `endpoints.empty()` or if the function object `c` returned `false` for all elements in the sequence.

(6.2) — `errc::operation_canceled` — if `s.is_open() == false` immediately following an `async_connect` operation on the underlying socket.

```
template<class Protocol, class InputIterator, class CompletionToken>
  DEDUCED async_connect(basic_socket<Protocol>& s,
                        InputIterator first, InputIterator last,
                        CompletionToken&& token);
```

7 *Returns:* `async_connect(s, first, last, [](auto, auto){ return true; }, forward<CompletionToken>(token))`.

```
template<class Protocol, class InputIterator,
  class ConnectCondition, class CompletionToken>
    DEDUCED async_connect(basic_socket<Protocol>& s,
                          InputIterator first, InputIterator last,
                          ConnectCondition c,
                          CompletionToken&& token);
```

8 A composed asynchronous operation (13.2.7.14).

9 *Completion signature:* `void(error_code ec, InputIterator i)`.

10 *Effects:* Performs `ec.clear()`, then finds the first iterator `i` in the range [`first`,`last`) for which:

(10.1) — `c(ec, *i)` yields `true`;

(10.2) — `s.close(ec)` succeeds;

(10.3) — `s.open(typename Protocol::endpoint(*i).protocol(), ec)` succeeds; and

(10.4) — the asynchronous operation `s.async_connect(*i, unspecified)` succeeds.

11 `ec` is updated with the result of the `s.async_connect(*i, unspecified)` operation, if any. If no such iterator is found, or if the operation fails with one of the error conditions listed below, `i` is set to `last`. [*Note:* The underlying `close`, `open`, and `async_connect` operations are performed sequentially. — *end note*]

12 *Error conditions:*

(12.1) — `socket_errc::not_found` — if `first == last` or if the function object `c` returned `false` for all iterators in the range.

(12.2) — `errc::operation_canceled` — if `s.is_open() == false` immediately following an `async_connect` operation on the underlying socket.

21 Internet protocol [internet]

21.1 Header <experimental/internet> synopsis [internet.synop]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  enum class resolver_errc {
    host_not_found = an implementation-defined non-zero value, // EAI_NONAME
    host_not_found_try_again = an implementation-defined non-zero value, // EAI_AGAIN
    service_not_found = an implementation-defined non-zero value // EAI_SERVICE
  };

  const error_category& resolver_category() noexcept;

  error_code make_error_code(resolver_errc e) noexcept;
  error_condition make_error_condition(resolver_errc e) noexcept;

  using port_type = uint_least16_t;
  using scope_id_type = uint_least32_t;

  struct v4_mapped_t {};
  constexpr v4_mapped_t v4_mapped;

  class address;
  class address_v4;
  class address_v6;

  class bad_address_cast;

  // address comparisons:
  constexpr bool operator==(const address&, const address&) noexcept;
  constexpr bool operator!=(const address&, const address&) noexcept;
  constexpr bool operator< (const address&, const address&) noexcept;
  constexpr bool operator> (const address&, const address&) noexcept;
  constexpr bool operator<=(const address&, const address&) noexcept;
  constexpr bool operator>=(const address&, const address&) noexcept;

  // address_v4 comparisons:
  constexpr bool operator==(const address_v4&, const address_v4&) noexcept;
  constexpr bool operator!=(const address_v4&, const address_v4&) noexcept;
  constexpr bool operator< (const address_v4&, const address_v4&) noexcept;
  constexpr bool operator> (const address_v4&, const address_v4&) noexcept;
  constexpr bool operator<=(const address_v4&, const address_v4&) noexcept;
  constexpr bool operator>=(const address_v4&, const address_v4&) noexcept;

  // address_v6 comparisons:
  constexpr bool operator==(const address_v6&, const address_v6&) noexcept;
  constexpr bool operator!=(const address_v6&, const address_v6&) noexcept;
```

```
constexpr bool operator< (const address_v6&, const address_v6&) noexcept;
constexpr bool operator> (const address_v6&, const address_v6&) noexcept;
constexpr bool operator<=(const address_v6&, const address_v6&) noexcept;
constexpr bool operator>=(const address_v6&, const address_v6&) noexcept;

// address creation:
address make_address(const char*);
address make_address(const char*, error_code&) noexcept;
address make_address(const string&);
address make_address(const string&, error_code&) noexcept;
address make_address(string_view);
address make_address(string_view, error_code&) noexcept;

// address_v4 creation:
constexpr address_v4 make_address_v4(const address_v4::bytes_type&);
constexpr address_v4 make_address_v4(address_v4::uint_type);
constexpr address_v4 make_address_v4(v4_mapped_t, const address_v6&);
address_v4 make_address_v4(const char*);
address_v4 make_address_v4(const char*, error_code&) noexcept;
address_v4 make_address_v4(const string&);
address_v4 make_address_v4(const string&, error_code&) noexcept;
address_v4 make_address_v4(string_view);
address_v4 make_address_v4(string_view, error_code&) noexcept;

// address_v6 creation:
constexpr address_v6 make_address_v6(const address_v6::bytes_type&,
                                     scope_id_type = 0);
constexpr address_v6 make_address_v6(v4_mapped_t, const address_v4&) noexcept;
address_v6 make_address_v6(const char*);
address_v6 make_address_v6(const char*, error_code&) noexcept;
address_v6 make_address_v6(const string&);
address_v6 make_address_v6(const string&, error_code&) noexcept;
address_v6 make_address_v6(string_view);
address_v6 make_address_v6(string_view, error_code&) noexcept;

// address I/O:
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>&, const address&);

// address_v4 I/O:
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>&, const address_v4&);

// address_v6 I/O:
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>&, const address_v6&);

template<class> class basic_address_iterator; // not defined
template<> class basic_address_iterator<address_v4>;
using address_v4_iterator = basic_address_iterator<address_v4>;
template<> class basic_address_iterator<address_v6>;
using address_v6_iterator = basic_address_iterator<address_v6>;
```

```
template<class> class basic_address_range; // not defined
template<> class basic_address_range<address_v4>;
using address_v4_range = basic_address_range<address_v4>;
template<> class basic_address_range<address_v6>;
using address_v6_range = basic_address_range<address_v6>;

class network_v4;
class network_v6;

// network_v4 comparisons:
bool operator==(const network_v4&, const network_v4&) noexcept;
bool operator!=(const network_v4&, const network_v4&) noexcept;

// network_v6 comparisons:
bool operator==(const network_v6&, const network_v6&) noexcept;
bool operator!=(const network_v6&, const network_v6&) noexcept;

// network_v4 creation:
network_v4 make_network_v4(const address_v4&, int);
network_v4 make_network_v4(const address_v4&, const address_v4&);
network_v4 make_network_v4(const char*);
network_v4 make_network_v4(const char*, error_code&) noexcept;
network_v4 make_network_v4(const string&);
network_v4 make_network_v4(const string&, error_code&) noexcept;
network_v4 make_network_v4(string_view);
network_v4 make_network_v4(string_view, error_code&) noexcept;

// network_v6 creation:
network_v6 make_network_v6(const address_v6&, int);
network_v6 make_network_v6(const char*);
network_v6 make_network_v6(const char*, error_code&) noexcept;
network_v6 make_network_v6(const string&);
network_v6 make_network_v6(const string&, error_code&) noexcept;
network_v6 make_network_v6(string_view);
network_v6 make_network_v6(string_view, error_code&) noexcept;

// network_v4 I/O:
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>&, const network_v4&);

// network_v6 I/O:
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>&, const network_v6&);

template<class InternetProtocol>
  class basic_endpoint;

// basic_endpoint comparisons:
template<class InternetProtocol>
  bool operator==(const basic_endpoint<InternetProtocol>&,
                  const basic_endpoint<InternetProtocol>&);
template<class InternetProtocol>
```

```
  bool operator!=(const basic_endpoint<InternetProtocol>&,
                  const basic_endpoint<InternetProtocol>&);
template<class InternetProtocol>
  bool operator< (const basic_endpoint<InternetProtocol>&,
                  const basic_endpoint<InternetProtocol>&);
template<class InternetProtocol>
  bool operator> (const basic_endpoint<InternetProtocol>&,
                  const basic_endpoint<InternetProtocol>&);
template<class InternetProtocol>
  bool operator<=(const basic_endpoint<InternetProtocol>&,
                  const basic_endpoint<InternetProtocol>&);
template<class InternetProtocol>
  bool operator>=(const basic_endpoint<InternetProtocol>&,
                  const basic_endpoint<InternetProtocol>&);

// basic_endpoint I/O:
template<class CharT, class Traits, class InternetProtocol>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>&,
    const basic_endpoint<InternetProtocol>&);

template<class InternetProtocol>
  class basic_resolver_entry;

template<class InternetProtocol>
  bool operator==(const basic_resolver_entry<InternetProtocol>&,
                  const basic_resolver_entry<InternetProtocol>&);
template<class InternetProtocol>
  bool operator!=(const basic_resolver_entry<InternetProtocol>&,
                  const basic_resolver_entry<InternetProtocol>&);

template<class InternetProtocol>
  class basic_resolver_results;

template<class InternetProtocol>
  bool operator==(const basic_resolver_results<InternetProtocol>&,
                  const basic_resolver_results<InternetProtocol>&);
template<class InternetProtocol>
  bool operator!=(const basic_resolver_results<InternetProtocol>&,
                  const basic_resolver_results<InternetProtocol>&);

class resolver_base;

template<class InternetProtocol>
  class basic_resolver;

string host_name();
string host_name(error_code&);
template<class Allocator>
  basic_string<char, char_traits<char>, Allocator>
    host_name(const Allocator&);
template<class Allocator>
  basic_string<char, char_traits<char>, Allocator>
    host_name(const Allocator&, error_code&);
```

```
    class tcp;

    // tcp comparisons:
    bool operator==(const tcp& a, const tcp& b);
    bool operator!=(const tcp& a, const tcp& b);

    class udp;

    // udp comparisons:
    bool operator==(const udp& a, const udp& b);
    bool operator!=(const udp& a, const udp& b);

    class v6_only;

    namespace unicast {

      class hops;

    } // namespace unicast

    namespace multicast {

      class join_group;

      class leave_group;

      class outbound_interface;

      class hops;

      class enable_loopback;

    } // namespace multicast
  } // namespace ip
  } // inline namespace v1
  } // namespace net
  } // namespace experimental

    template<> struct is_error_condition_enum<
      experimental::net::v1::ip::resolver_errc>
        : public true_type {};

    // hash support
    template<class T> struct hash;
    template<> struct hash<experimental::net::v1::ip::address>;
    template<> struct hash<experimental::net::v1::ip::address_v4>;
    template<> struct hash<experimental::net::v1::ip::address_v6>;

  } // namespace std
```

21.2 Requirements [internet.reqmts]

21.2.1 Internet protocol requirements [internet.reqmts.protocol]

1 A type `X` meets the `InternetProtocol` requirements if it satisfies the requirements of `AcceptableProtocol` (18.2.7), as well as the additional requirements listed below.

2 In the table below, `a` denotes a (possibly const) value of type `X`, and `b` denotes a (possibly const) value of type `X`.

Table 35 — InternetProtocol requirements

expression	return type	assertion/note pre/post-conditions
`X::resolver`	`ip::basic_-resolver<X>`	The type of a resolver for the protocol.
`X::v4()`	`X`	Returns an object representing the IP version 4 protocol.
`X::v6()`	`X`	Returns an object representing the IP version 6 protocol.
`a == b`	convertible to `bool`	Returns `true` if `a` and `b` represent the same IP protocol version, otherwise `false`.
`a != b`	convertible to `bool`	Returns `!(a == b)`.

21.2.2 Multicast group socket options [internet.reqmts.opt.mcast]

1 A type `X` meets the `MulticastGroupSocketOption` requirements if it satisfies the requirements of `Destructible` (C++Std [destructible]), `CopyConstructible` (C++Std [copyconstructible]), `CopyAssignable` (C++Std [copyassignable]), and `SettableSocketOption` (18.2.9), as well as the additional requirements listed below.

2 In the table below, `a` denotes a (possibly const) value of type `X`, `b` denotes a (possibly const) value of type `address`, `c` and `d` denote (possibly const) values of type `address_v4`, `e` denotes a (possibly const) value of type `address_v6`, `f` denotes a (possibly const) value of type `unsigned int`, and `u` denotes an identifier.

Table 36 — MulticastGroupSocketOption requirements

expression	type	assertion/note pre/post-conditions
`X u(b);`		Constructs a multicast group socket option to join the group with the specified version-independent address.
`X u(c, d);`		Constructs a multicast group socket option to join the specified IPv4 address on a specified network interface.
`X u(e, f);`		Constructs a multicast group socket option to join the specified IPv6 address on a specified network interface.

3 In this Technical Specification, types that satisfy the `MulticastGroupSocketOption` requirements are defined as follows.

```
class C
{
public:
  // constructors:
  explicit C(const address& multicast_group) noexcept;
  explicit C(const address_v4& multicast_group,
             const address_v4& network_interface = address_v4::any()) noexcept;
  explicit C(const address_v6& multicast_group,
             unsigned int network_interface = 0) noexcept;
};
```

4 Extensible implementations provide the following member functions:

```
class C
{
public:
  template<class Protocol> int level(const Protocol& p) const noexcept;
  template<class Protocol> int name(const Protocol& p) const noexcept;
  template<class Protocol> const void* data(const Protocol& p) const noexcept;
  template<class Protocol> size_t size(const Protocol& p) const noexcept;
  // remainder unchanged
private:
  ip_mreq v4_value_; // exposition only
  ipv6_mreq v6_value_; // exposition only
};
```

5 Let *L* and *N* identify the POSIX macros to be passed as the `level` and `option_name` arguments, respectively, to POSIX `setsockopt` and `getsockopt`.

```
explicit C(const address& multicast_group) noexcept;
```

6 *Effects:* If `multicast_group.is_v6()` is `true`, calls *C*`(multicast_group.to_v6())`; otherwise, calls *C*`(multicast_group.to_v4())`.

```
explicit C(const address_v4& multicast_group,
        const address_v4& network_interface = address_v4::any()) noexcept;
```

7 *Effects:* For extensible implementations, `v4_value_.imr_multiaddr` is initialized to correspond to the address `multicast_group`, `v4_value_.imr_interface` is initialized to correspond to address `network_interface`, and `v6_value_` is zero-initialized.

```
explicit C(const address_v6& multicast_group,
        unsigned int network_interface = 0) noexcept;
```

8 *Effects:* For extensible implementations, `v6_value_.ipv6mr_multiaddr` is initialized to correspond to the address `multicast_group`, `v6_value_.ipv6mr_interface` is initialized to `network_interface`, and `v4_value_` is zero-initialized.

```
template<class Protocol> int level(const Protocol& p) const noexcept;
```

9 *Returns: L.*

```
template<class Protocol> int name(const Protocol& p) const noexcept;
```

10 *Returns: N.*

```
template<class Protocol> const void* data(const Protocol& p) const noexcept;
```

11 *Returns:* `std::addressof(v6_value_)` if `p.family() == AF_INET6`, otherwise `std::addressof(v4_value_)`.

```
template<class Protocol> size_t size(const Protocol& p) const noexcept;
```

12 *Returns:* `sizeof(v6_value_)` if `p.family() == AF_INET6`, otherwise `sizeof(v4_value_)`.

21.3 Error codes [internet.resolver.err]

`const error_category& resolver_category() noexcept;`

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

2 The object's `default_error_condition` and `equivalent` virtual functions behave as specified for the class `error_category`. The object's `name` virtual function returns a pointer to the string `"resolver"`.

`error_code make_error_code(resolver_errc e) noexcept;`

3 *Returns:* `error_code(static_cast<int>(e), resolver_category())`.

`error_condition make_error_condition(resolver_errc e) noexcept;`

4 *Returns:* `error_condition(static_cast<int>(e), resolver_category())`.

21.4 Class `ip::address` [internet.address]

1 The class `address` is a version-independent representation for an IP address. An object of class `address` holds either an IPv4 address, an IPv6 address, or no valid address.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  class address
  {
  public:
    // constructors:
    constexpr address() noexcept;
    constexpr address(const address& a) noexcept;
    constexpr address(const address_v4& a) noexcept;
    constexpr address(const address_v6& a) noexcept;

    // assignment:
    address& operator=(const address& a) noexcept;
    address& operator=(const address_v4& a) noexcept;
    address& operator=(const address_v6& a) noexcept;

    // members:
    constexpr bool is_v4() const noexcept;
    constexpr bool is_v6() const noexcept;
    constexpr address_v4 to_v4() const;
    constexpr address_v6 to_v6() const;
    constexpr bool is_unspecified() const noexcept;
    constexpr bool is_loopback() const noexcept;
    constexpr bool is_multicast() const noexcept;
    template<class Allocator = allocator<char>>
      basic_string<char, char_traits<char>, Allocator>
        to_string(const Allocator& a = Allocator()) const;

  private:
    address_v4 v4_; // exposition only
    address_v6 v6_; // exposition only
```

```
  };

  // address comparisons:
  constexpr bool operator==(const address& a, const address& b) noexcept;
  constexpr bool operator!=(const address& a, const address& b) noexcept;
  constexpr bool operator< (const address& a, const address& b) noexcept;
  constexpr bool operator> (const address& a, const address& b) noexcept;
  constexpr bool operator<=(const address& a, const address& b) noexcept;
  constexpr bool operator>=(const address& a, const address& b) noexcept;

  // address creation:
  address make_address(const char* str);
  address make_address(const char* str, error_code& ec) noexcept;
  address make_address(const string& str);
  address make_address(const string& str, error_code& ec) noexcept;
  address make_address(string_view str);
  address make_address(string_view str, error_code& ec) noexcept;

  // address I/O:
  template<class CharT, class Traits>
    basic_ostream<CharT, Traits>& operator<<(
      basic_ostream<CharT, Traits>& os, const address& addr);

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 `address` satisfies the requirements for `Destructible` (C++Std [destructible]), `CopyConstructible` (C++Std [copyconstructible]), and `CopyAssignable` (C++Std [copyassignable]).

21.4.1 ip::address constructors [internet.address.cons]

`constexpr address() noexcept;`

1 *Postconditions:* `is_v4() == true`, `is_v6() == false`, and `is_unspecified() == true`.

`constexpr address(const address_v4& a) noexcept;`

2 *Effects:* Initializes `v4_` with `a`.

3 *Postconditions:* `is_v4() == true` and `is_v6() == false`.

`constexpr address(const address_v6& a) noexcept;`

4 *Effects:* Initializes `v6_` with `a`.

5 *Postconditions:* `is_v4() == false` and `is_v6() == true`.

21.4.2 ip::address assignment [internet.address.assign]

`address& operator=(const address_v4& a) noexcept;`

1 *Postconditions:* `is_v4() == true` and `is_v6() == false` and `to_v4() == a`.

2 *Returns:* `*this`

`address& operator=(const address_v6& a) noexcept;`

3 *Postconditions:* `is_v4() == false` and `is_v6() == true` and `to_v6() == a`.

4 *Returns:* `*this`

21.4.3 ip::address members [internet.address.members]

```
constexpr bool is_v4() const noexcept;
```

1 *Returns:* `true` if the object contains an IP version 4 address, otherwise `false`.

```
constexpr bool is_v6() const noexcept;
```

2 *Returns:* `true` if the object contains an IP version 6 address, otherwise `false`.

```
constexpr address_v4 to_v4() const;
```

3 *Returns:* `v4_`.

4 *Remarks:* `bad_address_cast` if `is_v4() == false`.

```
constexpr address_v6 to_v6() const;
```

5 *Returns:* `v6_`.

6 *Remarks:* `bad_address_cast` if `is_v6() == false`.

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

7 *Returns:* If `is_v4()`, returns `v4_.is_unspecified()`. Otherwise returns `v6_.is_unspecified()`.

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

8 *Returns:* If `is_v4()`, returns `v4_.is_loopback()`. Otherwise returns `v6_.is_loopback()`.

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

9 *Returns:* If `is_v4()`, returns `v4_.is_multicast()`. Otherwise returns `v6_.is_multicast()`.

```
template<class Allocator = allocator<char>>
  basic_string<char, char_traits<char>, Allocator>
    to_string(const Allocator& a = Allocator()) const;
```

10 *Returns:* If `is_v4()`, returns `v4_.to_string(a)`. Otherwise returns `v6_.to_string(a)`.

21.4.4 ip::address comparisons [internet.address.comparisons]

```
constexpr bool operator==(const address& a, const address& b) noexcept;
```

1 *Returns:*

(1.1) — if `a.is_v4() != b.is_v4()`, `false`;

(1.2) — if `a.is_v4()`, the result of `a.v4_ == b.v4_`;

(1.3) — otherwise, the result of `a.v6_ == b.v6_`.

```
constexpr bool operator!=(const address& a, const address& b) noexcept;
```

2 *Returns:* `!(a == b)`.

```
constexpr bool operator< (const address& a, const address& b) noexcept;
```

3 *Returns:*

(3.1) — if `a.is_v4() && !b.is_v4()`, `true`;

(3.2) — if `!a.is_v4() && b.is_v4()`, `false`;

(3.3) — if `a.is_v4()`, the result of `a.v4_ < b.v4_`;

(3.4) — otherwise, the result of `a.v6_ < b.v6_`.

```
constexpr bool operator> (const address& a, const address& b) noexcept;
```

4 *Returns:* `b < a`.

```
constexpr bool operator<=(const address& a, const address& b) noexcept;
```

5 *Returns:* `!(b < a)`.

```
constexpr bool operator>=(const address& a, const address& b) noexcept;
```

6 *Returns:* `!(a < b)`.

21.4.5 ip::address creation [internet.address.creation]

```
address make_address(const char* str);
address make_address(const char* str, error_code& ec) noexcept;
address make_address(const string& str);
address make_address(const string& str, error_code& ec) noexcept;
address make_address(string_view str);
address make_address(string_view str, error_code& ec) noexcept;
```

1 *Effects:* Converts a textual representation of an address into an object of class `address`, as if by calling:

```
address a;
address_v6 v6a = make_address_v6(str, ec);
if (!ec)
  a = v6a;
else
{
  address_v4 v4a = make_address_v4(str, ec);
  if (!ec)
    a = v4a;
}
```

2 *Returns:* `a`.

21.4.6 ip::address I/O [internet.address.io]

```
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os, const address& addr);
```

1 *Returns:* `os << addr.to_string().c_str()`.

21.5 Class ip::address_v4 [internet.address.v4]

1 The class `address_v4` is a representation of an IPv4 address.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
```

```
namespace ip {

  class address_v4
  {
  public:
    // types:
    using uint_type = uint_least32_t;
    struct bytes_type;

    // constructors:
    constexpr address_v4() noexcept;
    constexpr address_v4(const address_v4& a) noexcept;
    constexpr address_v4(const bytes_type& bytes);
    explicit constexpr address_v4(uint_type val);

    // assignment:
    address_v4& operator=(const address_v4& a) noexcept;

    // members:
    constexpr bool is_unspecified() const noexcept;
    constexpr bool is_loopback() const noexcept;
    constexpr bool is_multicast() const noexcept;
    constexpr bytes_type to_bytes() const noexcept;
    constexpr uint_type to_uint() const noexcept;
    template<class Allocator = allocator<char>>
      basic_string<char, char_traits<char>, Allocator>
        to_string(const Allocator& a = Allocator()) const;

    // static members:
    static constexpr address_v4 any() noexcept;
    static constexpr address_v4 loopback() noexcept;
    static constexpr address_v4 broadcast() noexcept;
  };

  // address_v4 comparisons:
  constexpr bool operator==(const address_v4& a, const address_v4& b) noexcept;
  constexpr bool operator!=(const address_v4& a, const address_v4& b) noexcept;
  constexpr bool operator< (const address_v4& a, const address_v4& b) noexcept;
  constexpr bool operator> (const address_v4& a, const address_v4& b) noexcept;
  constexpr bool operator<=(const address_v4& a, const address_v4& b) noexcept;
  constexpr bool operator>=(const address_v4& a, const address_v4& b) noexcept;

  // address_v4 creation:
  constexpr address_v4 make_address_v4(const address_v4::bytes_type& bytes);
  constexpr address_v4 make_address_v4(address_v4::uint_type val);
  constexpr address_v4 make_address_v4(v4_mapped_t, const address_v6& a);
  address_v4 make_address_v4(const char* str);
  address_v4 make_address_v4(const char* str, error_code& ec) noexcept;
  address_v4 make_address_v4(const string& str);
  address_v4 make_address_v4(const string& str, error_code& ec) noexcept;
  address_v4 make_address_v4(string_view str);
  address_v4 make_address_v4(string_view str, error_code& ec) noexcept;

  // address_v4 I/O:
  template<class CharT, class Traits>
```

```
      basic_ostream<CharT, Traits>& operator<<(
        basic_ostream<CharT, Traits>& os, const address_v4& addr);

  } // namespace ip
  } // inline namespace v1
  } // namespace net
  } // namespace experimental
  } // namespace std
```

2 address_v4 satisfies the requirements for Destructible (C++Std [destructible]), CopyConstructible (C++Std [copyconstructible]), and CopyAssignable (C++Std [copyassignable]).

21.5.1 Struct ip::address_v4::bytes_type [internet.address.v4.bytes]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  struct address_v4::bytes_type : array<unsigned char, 4>
  {
    template<class... T> explicit constexpr bytes_type(T... t)
      : array<unsigned char, 4>{{static_cast<unsigned char>(t)...}} {}
  };

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 The ip::address_v4::bytes_type type is a standard-layout struct that provides a byte-level representation of an IPv4 address in network byte order.

21.5.2 ip::address_v4 constructors [internet.address.v4.cons]

```
constexpr address_v4() noexcept;
```

1 *Postconditions:* to_bytes() yields {0, 0, 0, 0} and to_uint() == 0.

```
constexpr address_v4(const bytes_type& bytes);
```

2 *Remarks:* out_of_range if any element of bytes is not in the range [0, 0xFF]. [*Note:* For implementations where numeric_limits<unsigned char>::max() == 0xFF, no out-of-range detection is needed. — *end note*]

3 *Postconditions:* to_bytes() == bytes and to_uint() == (bytes[0] << 24) | (bytes[1] << 16) | (bytes[2] << 8) | bytes[3].

```
explicit constexpr address_v4(address_v4::uint_type val);
```

4 *Remarks:* out_of_range if val is not in the range [0, 0xFFFFFFFF]. [*Note:* For implementations where numeric_limits<address_v4::uint_type>::max() == 0xFFFFFFFF, no out-of-range detection is needed. — *end note*]

5 *Postconditions:* to_uint() == val and to_bytes() is { (val >> 24) & 0xFF, (val >> 16) & 0xFF, (val >> 8) & 0xFF, val & 0xFF }.

21.5.3 `ip::address_v4` members [internet.address.v4.members]

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

1 *Returns:* `to_uint() == 0`.

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

2 *Returns:* `(to_uint() & 0xFF000000) == 0x7F000000`.

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

3 *Returns:* `(to_uint() & 0xF0000000) == 0xE0000000`.

```
constexpr bytes_type to_bytes() const noexcept;
```

4 *Returns:* A representation of the address in network byte order (3.2).

```
constexpr address_v4::uint_type to_uint() const noexcept;
```

5 *Returns:* A representation of the address in host byte order (3.1).

```
template<class Allocator = allocator<char>>
  basic_string<char, char_traits<char>, Allocator>
    to_string(const Allocator& a = Allocator()) const;
```

6 *Returns:* If successful, the textual representation of the address, determined as if by POSIX `inet_ntop` when invoked with address family `AF_INET`. Otherwise `basic_string<char, char_traits<char>, Allocator>(a)`.

21.5.4 `ip::address_v4` static members [internet.address.v4.static]

```
static constexpr address_v4 any() noexcept;
```

1 *Returns:* `address_v4()`.

```
static constexpr address_v4 loopback() noexcept;
```

2 *Returns:* `address_v4(0x7F000001)`.

```
static constexpr address_v4 broadcast() noexcept;
```

3 *Returns:* `address_v4(0xFFFFFFFF)`.

21.5.5 `ip::address_v4` comparisons [internet.address.v4.comparisons]

```
constexpr bool operator==(const address_v4& a, const address_v4& b) noexcept;
```

1 *Returns:* `a.to_uint() == b.to_uint()`.

```
constexpr bool operator!=(const address_v4& a, const address_v4& b) noexcept;
```

2 *Returns:* `!(a == b)`.

```
constexpr bool operator< (const address_v4& a, const address_v4& b) noexcept;
```

3 *Returns:* `a.to_uint() < b.to_uint()`.

```
constexpr bool operator> (const address_v4& a, const address_v4& b) noexcept;
```

4 *Returns:* `b < a`.

```
constexpr bool operator<=(const address_v4& a, const address_v4& b) noexcept;
```

5 *Returns:* `!(b < a)`.

```
constexpr bool operator>=(const address_v4& a, const address_v4& b) noexcept;
```

6 *Returns:* `!(a < b)`.

21.5.6 ip::address_v4 creation [internet.address.v4.creation]

```
constexpr address_v4 make_address_v4(const address_v4::bytes_type& bytes);
```

1 *Returns:* `address_v4(bytes)`.

```
constexpr address_v4 make_address_v4(address_v4::uint_type val);
```

2 *Returns:* `address_v4(val)`.

```
constexpr address_v4 make_address_v4(v4_mapped_t, const address_v6& a);
```

3 *Returns:* An `address_v4` object corresponding to the IPv4-mapped IPv6 address, as if computed by the following method:

```
address_v6::bytes_type v6b = a.to_bytes();
address_v4::bytes_type v4b(v6b[12], v6b[13], v6b[14], v6b[15]);
return address_v4(v4b);
```

4 *Remarks:* `bad_address_cast` if `a.is_v4_mapped()` is `false`.

```
address_v4 make_address_v4(const char* str);
address_v4 make_address_v4(const char* str, error_code& ec) noexcept;
address_v4 make_address_v4(const string& str);
address_v4 make_address_v4(const string& str, error_code& ec) noexcept;
address_v4 make_address_v4(string_view str);
address_v4 make_address_v4(string_view str, error_code& ec) noexcept;
```

5 *Effects:* Converts a textual representation of an address into a corresponding `address_v4` value, as if by POSIX `inet_pton` when invoked with address family `AF_INET`.

6 *Returns:* If successful, an `address_v4` value corresponding to the string `str`. Otherwise `address_v4()`.

7 *Error conditions:*

(7.1) — `errc::invalid_argument` — if `str` is not a valid textual representation of an IPv4 address.

21.5.7 ip::address_v4 I/O [internet.address.v4.io]

```
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os, const address_v4& addr);
```

1 *Returns:* `os << addr.to_string().c_str()`.

21.6 Class ip::address_v6 [internet.address.v6]

1 The class `address_v6` is a representation of an IPv6 address.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {
```

```
class address_v6
{
public:
  // types:
  struct bytes_type;

  // constructors:
  constexpr address_v6() noexcept;
  constexpr address_v6(const address_v6& a) noexcept;
  constexpr address_v6(const bytes_type& bytes,
                       scope_id_type scope = 0);

  // assignment:
  address_v6& operator=(const address_v6& a) noexcept;

  // members:
  void scope_id(scope_id_type id) noexcept;
  constexpr scope_id_type scope_id() const noexcept;
  constexpr bool is_unspecified() const noexcept;
  constexpr bool is_loopback() const noexcept;
  constexpr bool is_multicast() const noexcept;
  constexpr bool is_link_local() const noexcept;
  constexpr bool is_site_local() const noexcept;
  constexpr bool is_v4_mapped() const noexcept;
  constexpr bool is_multicast_node_local() const noexcept;
  constexpr bool is_multicast_link_local() const noexcept;
  constexpr bool is_multicast_site_local() const noexcept;
  constexpr bool is_multicast_org_local() const noexcept;
  constexpr bool is_multicast_global() const noexcept;
  constexpr bytes_type to_bytes() const noexcept;
  template<class Allocator = allocator<char>>
    basic_string<char, char_traits<char>, Allocator>
      to_string(const Allocator& a = Allocator()) const;

  // static members:
  static constexpr address_v6 any() noexcept;
  static constexpr address_v6 loopback() noexcept;
};

// address_v6 comparisons:
constexpr bool operator==(const address_v6& a, const address_v6& b) noexcept;
constexpr bool operator!=(const address_v6& a, const address_v6& b) noexcept;
constexpr bool operator< (const address_v6& a, const address_v6& b) noexcept;
constexpr bool operator> (const address_v6& a, const address_v6& b) noexcept;
constexpr bool operator<=(const address_v6& a, const address_v6& b) noexcept;
constexpr bool operator>=(const address_v6& a, const address_v6& b) noexcept;

// address_v6 creation:
constexpr address_v6 make_address_v6(const address_v6::bytes_type& bytes,
                                     scope_id_type scope_id = 0);
constexpr address_v6 make_address_v6(v4_mapped_t, const address_v4& a) noexcept;
address_v6 make_address_v6(const char* str);
address_v6 make_address_v6(const char* str, error_code& ec) noexcept;
address_v6 make_address_v6(const string& str);
```

```
  address_v6 make_address_v6(const string& str, error_code& ec) noexcept;
  address_v6 make_address_v6(string_view str);
  address_v6 make_address_v6(string_view str, error_code& ec) noexcept;

  // address_v6 I/O:
  template<class CharT, class Traits>
    basic_ostream<CharT, Traits>& operator<<(
      basic_ostream<CharT, Traits>& os, const address_v6& addr);

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 `address_v6` satisfies the requirements for `Destructible` (C++Std [destructible]), `CopyConstructible` (C++Std [copyconstructible]), and `CopyAssignable` (C++Std [copyassignable]).

3 [*Note:* The implementations of the functions `is_unspecified`, `is_loopback`, `is_multicast`, `is_link_local`, `is_site_local`, `is_v4_mapped`, `is_multicast_node_local`, `is_multicast_link_local`, `is_multicast_site_local`, `is_multicast_org_local` and `is_multicast_global` are determined by [RFC4291]. — *end note*]

21.6.1 Struct `ip::address_v6::bytes_type` [internet.address.v6.bytes]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  struct address_v6::bytes_type : array<unsigned char, 16>
  {
    template<class... T> explicit constexpr bytes_type(T... t)
      : array<unsigned char, 16>{{static_cast<unsigned char>(t)...}} {}
  };

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 The `ip::address_v6::bytes_type` type is a standard-layout struct that provides a byte-level representation of an IPv6 address in network byte order.

21.6.2 `ip::address_v6` constructors [internet.address.v6.cons]

```
constexpr address_v6() noexcept;
```

1 *Postconditions:* `is_unspecified() == true` and `scope_id() == 0`.

```
constexpr address_v6(const bytes_type& bytes,
                     scope_id_type scope = 0);
```

2 *Remarks:* `out_of_range` if any element of `bytes` is not in the range `[0, 0xFF]`. [*Note:* For implementations where `numeric_limits<unsigned char>::max() == 0xFF`, no out-of-range detection is needed. — *end note*]

3 *Postconditions:* `to_bytes() == bytes` and `scope_id() == scope`.

21.6.3 ip::address_v6 members [internet.address.v6.members]

```
void scope_id(scope_id_type id) noexcept;
```

1 *Postconditions:* `scope_id() == id`.

```
constexpr scope_id_type scope_id() const noexcept;
```

2 *Returns:* The scope identifier associated with the address.

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

3 *Returns:* `*this == make_address_v6("::")`.

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

4 *Returns:* `*this == make_address_v6("::1")`.

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

5 *Returns:* A boolean indicating whether the `address_v6` object represents a multicast address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFF;
```

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

6 *Returns:* A boolean indicating whether the `address_v6` object represents a unicast link-local address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFE && (b[1] & 0xC0) == 0x80;
```

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

7 *Returns:* A boolean indicating whether the `address_v6` object represents a unicast site-local address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[0] == 0xFE && (b[1] & 0xC0) == 0xC0;
```

```
constexpr bool is_v4_mapped() const noexcept;
```

8 *Returns:* A boolean indicating whether the `address_v6` object represents an IPv4-mapped IPv6 address, as if computed by the following method:

```
bytes_type b = to_bytes();
return b[ 0] == 0 && b[ 1] == 0 && b[ 2] == 0    && b[ 3] == 0
    && b[ 4] == 0 && b[ 5] == 0 && b[ 6] == 0    && b[ 7] == 0
    && b[ 8] == 0 && b[ 9] == 0 && b[10] == 0xFF && b[11] == 0xFF;
```

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

9 *Returns:* `is_multicast() && (to_bytes()[1] & 0x0F) == 0x01`.

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

10 *Returns:* is_multicast() && (to_bytes()[1] & 0x0F) == 0x02.

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

11 *Returns:* is_multicast() && (to_bytes()[1] & 0x0F) == 0x05.

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

12 *Returns:* is_multicast() && (to_bytes()[1] & 0x0F) == 0x08.

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

13 *Returns:* is_multicast() && (to_bytes()[1] & 0x0F) == 0x0E.

```
constexpr bytes_type to_bytes() const noexcept;
```

14 *Returns:* A representation of the address in network byte order (3.2).

```
template<class Allocator = allocator<char>>
  basic_string<char, char_traits<char>, Allocator>
    to_string(const Allocator& a = Allocator()) const;
```

15 *Effects:* Converts an address into a textual representation. If scope_id() == 0, converts as if by POSIX inet_ntop when invoked with address family AF_INET6. If scope_id() != 0, the format is address%scope-id, where address is the textual representation of the equivalent address having scope_id() == 0, and scope-id is an implementation-defined textual representation of the scope identifier.

16 *Returns:* If successful, the textual representation of the address. Otherwise basic_string<char, char_traits<char>, Allocator>(a).

21.6.4 ip::address_v6 static members [internet.address.v6.static]

```
static constexpr address_v6 any() noexcept;
```

1 *Returns:* An address a such that the a.is_unspecified() == true and a.scope_id() == 0.

```
static constexpr address_v6 loopback() noexcept;
```

2 *Returns:* An address a such that the a.is_loopback() == true and a.scope_id() == 0.

21.6.5 ip::address_v6 comparisons [internet.address.v6.comparisons]

```
constexpr bool operator==(const address_v6& a, const address_v6& b) noexcept;
```

1 *Returns:* a.to_bytes() == b.to_bytes() && a.scope_id() == b.scope_id().

```
constexpr bool operator!=(const address_v6& a, const address_v6& b) noexcept;
```

2 *Returns:* !(a == b).

```
constexpr bool operator< (const address_v6& a, const address_v6& b) noexcept;
```

3 *Returns:* a.to_bytes() < b.to_bytes() || (!(b.to_bytes() < a.to_bytes()) && a.scope_id() < b.scope_id()).

```
constexpr bool operator> (const address_v6& a, const address_v6& b) noexcept;
```

4 *Returns:* b < a.

```
constexpr bool operator<=(const address_v6& a, const address_v6& b) noexcept;
```

5 *Returns:* !(b < a).

```
constexpr bool operator>=(const address_v6& a, const address_v6& b) noexcept;
```

6 *Returns:* !(a < b).

21.6.6 ip::address_v6 creation [internet.address.v6.creation]

```
constexpr address_v6 make_address_v6(const address_v6::bytes_type& bytes,
                                     scope_id_type scope_id);
```

1 *Returns:* `address_v6(bytes, scope_id)`.

```
constexpr address_v6 make_address_v6(v4_mapped_t, const address_v4& a) noexcept;
```

2 *Returns:* An `address_v6` object containing the IPv4-mapped IPv6 address corresponding to the specified IPv4 address, as if computed by the following method:

```
address_v4::bytes_type v4b = a.to_bytes();
address_v6::bytes_type v6b(0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                           0xFF, 0xFF, v4b[0], v4b[1], v4b[2], v4b[3]);
return address_v6(v6b);
```

```
address_v6 make_address_v6(const char* str);
address_v6 make_address_v6(const char* str, error_code& ec) noexcept;
address_v4 make_address_v6(const string& str);
address_v4 make_address_v6(const string& str, error_code& ec) noexcept;
address_v6 make_address_v6(string_view str);
address_v6 make_address_v6(string_view str, error_code& ec) noexcept;
```

3 *Effects:* Converts a textual representation of an address into a corresponding `address_v6` value. The format is either `address` or `address%scope-id`, where `address` is in the format specified by POSIX `inet_pton` when invoked with address family `AF_INET6`, and `scope-id` is an optional string specifying the scope identifier. All implementations accept as `scope-id` a textual representation of an unsigned decimal integer. It is implementation-defined whether alternative scope identifier representations are permitted. If `scope-id` is not supplied, an `address_v6` object is returned such that `scope_id() == 0`.

4 *Returns:* If successful, an `address_v6` value corresponding to the string `str`. Otherwise returns `address_v6()`.

5 *Error conditions:*

(5.1) — `errc::invalid_argument` — if `str` is not a valid textual representation of an IPv6 address.

21.6.7 ip::address_v6 I/O [internet.address.v6.io]

```
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os, const address_v6& addr);
```

1 *Returns:* `os << addr.to_string().c_str()`.

21.7 Class ip::bad_address_cast [internet.bad.address.cast]

1 An exception of type `bad_address_cast` is thrown by a failed `address_cast`.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  class bad_address_cast : public bad_cast
  {
  public:
```

```
    // constructor:
    bad_address_cast() noexcept;
  };

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

```
bad_address_cast() noexcept;
```

2 *Effects:* constructs a `bad_address_cast` object.

3 *Postconditions:* `what()` returns an implementation-defined NTBS.

21.8 Hash support [internet.hash]

```
template<> struct hash<experimental::net::v1::ip::address>;
template<> struct hash<experimental::net::v1::ip::address_v4>;
template<> struct hash<experimental::net::v1::ip::address_v6>;
```

1 *Requires:* the template specializations shall meet the requirements of class template `hash` (C++Std [unord.hash]).

21.9 Class template `ip::basic_address_iterator` specializations[internet.address.iter]

1 The class template `basic_address_iterator` enables iteration over IP addresses in network byte order. This clause defines two specializations of the class template `basic_address_iterator`: `basic_address_iterator<address_v4>` and `basic_address_iterator<address_v6>`. The members and operational semantics of these specializations are defined below.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  template<> class basic_address_iterator<Address>
  {
  public:
    // types:
    using value_type = Address;
    using difference_type = ptrdiff_t;
    using pointer = const Address*;
    using reference = const Address&;
    using iterator_category = input_iterator_tag;

    // constructors:
    basic_address_iterator(const Address& a) noexcept;

    // members:
    reference operator*() const noexcept;
    pointer operator->() const noexcept;
    basic_address_iterator& operator++() noexcept;
    basic_address_iterator operator++(int) noexcept;
    basic_address_iterator& operator--() noexcept;
```

```
    basic_address_iterator operator--(int) noexcept;

    // other members as required by C++Std [input.iterators]

  private:
    Address address_; // exposition only
  };

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Specializations of `basic_address_iterator` satisfy the requirements for input iterators (C++Std [input.iterators]).

```
basic_address_iterator(const Address& a) noexcept;
```

3 *Effects:* Initializes `address_` with `a`.

```
reference operator*() const noexcept;
```

4 *Returns:* `address_`.

```
pointer operator->() const noexcept;
```

5 *Returns:* `std::addressof(address_)`.

```
basic_address_iterator& operator++() noexcept;
```

6 *Effects:* Sets `address_` to the next unique address in network byte order.

7 *Returns:* `*this`.

```
basic_address_iterator operator++(int) noexcept;
```

8 *Effects:* Sets `address_` to the next unique address in network byte order.

9 *Returns:* The prior value of `*this`.

```
basic_address_iterator& operator--() noexcept;
```

10 *Effects:* Sets `address_` to the prior unique address in network byte order.

11 *Returns:* `*this`.

```
basic_address_iterator operator--(int) noexcept;
```

12 *Effects:* Sets `address_` to the prior unique address in network byte order.

13 *Returns:* The prior value of `*this`.

21.10 Class template `ip::basic_address_range` specializations [internet.address.range]

1 The class template `basic_address_range` represents a range of IP addresses in network byte order. This clause defines two specializations of the class template `basic_address_range`: `basic_address_range<address_v4>` and `basic_address_range<address_v6>`. The members and operational semantics of these specializations are defined below.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  template<> class basic_address_range<Address>
  {
  public:
    // types:
    using iterator = basic_address_iterator<Address>;

    // constructors:
    basic_address_range() noexcept;
    basic_address_range(const Address& first,
                        const Address& last) noexcept;

    // members:
    iterator begin() const noexcept;
    iterator end() const noexcept;
    bool empty() const noexcept;
    size_t size() const noexcept; // not always defined
    iterator find(const Address& addr) const noexcept;
  };

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Specializations of `basic_address_range` satisfy the requirements for `Destructible` (C++Std [destructible]), `CopyConstructible` (C++Std [copyconstructible]), and `CopyAssignable` (C++Std [copyassignable]).

```
basic_address_range() noexcept;
```

3 *Effects:* Constructs an object of type `basic_address_range<Address>` that represents an empty range.

```
basic_address_range(const Address& first,
                    const Address& last) noexcept;
```

4 *Effects:* Constructs an object of type `basic_address_range<Address>` that represents the half-open range `[first, last)`.

```
iterator begin() const noexcept;
```

5 *Returns:* An iterator that points to the beginning of the range.

```
iterator end() const noexcept;
```

6 *Returns:* An iterator that points to the end of the range.

```
bool empty() const noexcept;
```

7 *Returns:* `true` if `*this` represents an empty range, otherwise `false`.

```
size_t size() const noexcept;
```

8 *Returns:* The number of unique addresses in the range.

9 *Remarks:* This member function is not defined when *Address* is type `address_v6`.

`iterator find(const Address& addr) const noexcept;`

10 *Returns:* If `addr` is in the range, an iterator that points to `addr`; otherwise, `end()`.

11 Complexity: Constant time.

21.11 Class template `ip::network_v4` [internet.network.v4]

1 The class `network_v4` provides the ability to use and manipulate IPv4 network addresses.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  class network_v4
  {
  public:
    // constructors:
    constexpr network_v4() noexcept;
    constexpr network_v4(const address_v4& addr, int prefix_len);
    constexpr network_v4(const address_v4& addr, const address_v4& mask);

    // members:
    constexpr address_v4 address() const noexcept;
    constexpr int prefix_length() const noexcept;
    constexpr address_v4 netmask() const noexcept;
    constexpr address_v4 network() const noexcept;
    constexpr address_v4 broadcast() const noexcept;
    address_v4_range hosts() const noexcept;
    constexpr network_v4 canonical() const noexcept;
    constexpr bool is_host() const noexcept;
    constexpr bool is_subnet_of(const network_v4& other) const noexcept;
    template<class Allocator = allocator<char>>
      basic_string<char, char_traits<char>, Allocator>
        to_string(const Allocator& a = Allocator()) const;
  };

  // network_v4 comparisons:
  constexpr bool operator==(const network_v4& a, const network_v4& b) noexcept;
  constexpr bool operator!=(const network_v4& a, const network_v4& b) noexcept;

  // network_v4 creation:
  constexpr network_v4 make_network_v4(const address_v4& addr, int prefix_len);
  constexpr network_v4 make_network_v4(const address_v4& addr, const address_v4& mask);
  network_v4 make_network_v4(const char* str);
  network_v4 make_network_v4(const char* str, error_code& ec) noexcept;
  network_v4 make_network_v4(const string& str);
  network_v4 make_network_v4(const string& str, error_code& ec) noexcept;
  network_v4 make_network_v4(string_view str);
  network_v4 make_network_v4(string_view str, error_code& ec) noexcept;

  // network_v4 I/O:
```

```
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os, const network_v4& net);

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 `network_v4` satisfies the requirements for `Destructible` (C++Std [destructible]), `CopyConstructible` (C++Std [copyconstructible]), and `CopyAssignable` (C++Std [copyassignable]).

21.11.1 ip::network_v4 constructors [internet.network.v4.cons]

`constexpr network_v4() noexcept;`

1 *Postconditions:* `this->address().is_unspecified() == true` and `prefix_length() == 0`.

`constexpr network_v4(const address_v4& addr, int prefix_len);`

2 *Postconditions:* `this->address() == addr` and `prefix_length() == prefix_len`.

3 *Remarks:* `out_of_range` if `prefix_len < 0` or `prefix_len > 32`.

`constexpr network_v4(const address_v4& addr, const address_v4& mask);`

4 *Postconditions:* `this->address() == addr` and `prefix_length()` is equal to the number of contiguous non-zero bits in `mask`.

5 *Remarks:* `invalid_argument` if `mask` contains non-contiguous non-zero bits, or if the most significant bit is zero and any other bits are non-zero.

21.11.2 ip::network_v4 members [internet.network.v4.members]

`constexpr address_v4 address() const noexcept;`

1 *Returns:* The address specified when the `network_v4` object was constructed.

`constexpr int prefix_length() const noexcept;`

2 *Returns:* The prefix length of the network.

`constexpr address_v4 netmask() const noexcept;`

3 *Returns:* An `address_v4` object with `prefix_length()` contiguous non-zero bits set, starting from the most significant bit in network byte order. All other bits are zero.

`constexpr address_v4 network() const noexcept;`

4 *Returns:* An `address_v4` object with the first `prefix_length()` bits, starting from the most significant bit in network byte order, set to the corresponding bit value of `this->address()`. All other bits are zero.

`constexpr address_v4 broadcast() const noexcept;`

5 *Returns:* An `address_v4` object with the first `prefix_length()` bits, starting from the most significant bit in network byte order, set to the corresponding bit value of `this->address()`. All other bits are non-zero.

```
address_v4_range hosts() const noexcept;
```

6 *Returns:* If `is_host() == true`, an `address_v4_range` object representing the single address `this->address()`. Otherwise, an `address_v4_range` object representing the range of unique host IP addresses in the network.

7 [*Note:* For IPv4, the network address and the broadcast address are not included in the range of host IP addresses. For example, given a network `192.168.1.0/24`, the range returned by `hosts()` is from `192.168.1.1` to `192.168.1.254` inclusive, and neither `192.168.1.0` nor the broadcast address `192.168.1.255` are in the range. — *end note*]

```
constexpr network_v4 canonical() const noexcept;
```

8 *Returns:* `network_v4(network(), prefix_length())`.

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

9 *Returns:* `prefix_length() == 32`.

```
constexpr bool is_subnet_of(const network_v4& other) const noexcept;
```

10 *Returns:* `true` if `other.prefix_length() < prefix_length()` and `network_v4(this->address(), other.prefix_length()).canonical() == other.canonical()`, otherwise `false`.

```
template<class Allocator = allocator<char>>
  basic_string<char, char_traits<char>, Allocator>
    to_string(const Allocator& a = Allocator()) const;
```

11 *Returns:* `this->address().to_string(a) + "/" + std::to_string(prefix_length())`.

21.11.3 ip::network_v4 comparisons [internet.network.v4.comparisons]

```
constexpr bool operator==(const network_v4& a, const network_v4& b) noexcept;
```

1 *Returns:* `true` if `a.address() == b.address()` and `a.prefix_length() == b.prefix_length()`, otherwise `false`.

```
constexpr bool operator!=(const network_v4& a, const network_v4& b) noexcept;
```

2 *Returns:* `!(a == b)`.

21.11.4 ip::network_v4 creation [internet.network.v4.creation]

```
constexpr network_v4 make_network_v4(const address_v4& addr, int prefix_len);
```

1 *Returns:* `network_v4(addr, prefix_len)`.

```
constexpr network_v4 make_network_v4(const address_v4& addr, const address_v4& mask);
```

2 *Returns:* `network_v4(addr, mask)`.

```
network_v4 make_network_v4(const char* str);
network_v4 make_network_v4(const char* str, error_code& ec) noexcept;
network_v4 make_network_v4(const string& str);
network_v4 make_network_v4(const string& str, error_code& ec) noexcept;
network_v4 make_network_v4(string_view str);
network_v4 make_network_v4(string_view str, error_code& ec) noexcept;
```

3 *Returns:* If `str` contains a value of the form address `'/'` prefix-length, a `network_v4` object constructed with the result of applying `make_address_v4()` to the address portion of the string, and the result of converting prefix-length to an integer of type `int`. Otherwise returns `network_v4()` and sets `ec` to reflect the error.

4 *Error conditions:*

(4.1) — `errc::invalid_argument` — if `str` is not a valid textual representation of an IPv4 address and prefix length.

21.11.5 `ip::network_v4` I/O [internet.network.v4.io]

```
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os, const network_v4& net);
```

1 *Returns:* `os << net.to_string().c_str()`.

21.12 Class template `ip::network_v6` [internet.network.v6]

1 The class `network_v6` provides the ability to use and manipulate IPv6 network addresses.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  class network_v6
  {
  public:
    // constructors:
    constexpr network_v6() noexcept;
    constexpr network_v6(const address_v6& addr, int prefix_len);

    // members:
    constexpr address_v6 address() const noexcept;
    constexpr int prefix_length() const noexcept;
    constexpr address_v6 network() const noexcept;
    address_v6_range hosts() const noexcept;
    constexpr network_v6 canonical() const noexcept;
    constexpr bool is_host() const noexcept;
    constexpr bool is_subnet_of(const network_v6& other) const noexcept;
    template<class Allocator = allocator<char>>
      basic_string<char, char_traits<char>, Allocator>
        to_string(const Allocator& a = Allocator()) const;
  };

  // network_v6 comparisons:
  constexpr bool operator==(const network_v6& a, const network_v6& b) noexcept;
  constexpr bool operator!=(const network_v6& a, const network_v6& b) noexcept;

  // network_v6 creation:
  constexpr network_v6 make_network_v6(const address_v6& addr, int prefix_len);
  network_v6 make_network_v6(const char* str);
  network_v6 make_network_v6(const char* str, error_code& ec) noexcept;
  network_v6 make_network_v6(const string& str);
```

```
  network_v6 make_network_v6(const string& str, error_code& ec) noexcept;
  network_v6 make_network_v6(string_view str);
  network_v6 make_network_v6(string_view str, error_code& ec) noexcept;

  // network_v6 I/O:
  template<class CharT, class Traits>
    basic_ostream<CharT, Traits>& operator<<(
      basic_ostream<CharT, Traits>& os, const network_v6& net);

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 `network_v6` satisfies the requirements for `Destructible` (C++Std [destructible]), `CopyConstructible` (C++Std [copyconstructible]), and `CopyAssignable` (C++Std [copyassignable]).

21.12.1 ip::network_v6 constructors [internet.network.v6.cons]

`constexpr network_v6() noexcept;`

1 *Postconditions:* `this->address().is_unspecified() == true` and `prefix_length() == 0`.

`constexpr network_v6(const address_v6& addr, int prefix_len);`

2 *Postconditions:* `this->address() == addr` and `prefix_length() == prefix_len`.

3 *Remarks:* `out_of_range` if `prefix_len < 0` or `prefix_len > 128`.

21.12.2 ip::network_v6 members [internet.network.v6.members]

`constexpr address_v6 address() const noexcept;`

1 *Returns:* The address specified when the `network_v6` object was constructed.

`constexpr int prefix_length() const noexcept;`

2 *Returns:* The prefix length of the network.

`constexpr address_v6 network() const noexcept;`

3 *Returns:* An `address_v6` object with the first `prefix_length()` bits, starting from the most significant bit in network byte order, set to the corresponding bit value of `this->address()`. All other bits are zero.

`address_v6_range hosts() const noexcept;`

4 *Returns:* If `is_host() == true`, an `address_v6_range` object representing the single address `this->address()`. Otherwise, an `address_v6_range` object representing the range of unique host IP addresses in the network.

`constexpr network_v6 canonical() const noexcept;`

5 *Returns:* `network_v6(network(), prefix_length())`.

`constexpr bool is_host() const noexcept;`

6 *Returns:* `prefix_length() == 128`.

```
constexpr bool is_subnet_of(const network_v6& other) const noexcept;
```

7 *Returns:* `true` if `other.prefix_length() < prefix_length()` and `network_v6(this->address(), other.prefix_length()).canonical() == other.canonical()`, otherwise `false`.

```
template<class Allocator = allocator<char>>
  basic_string<char, char_traits<char>, Allocator>
    to_string(const Allocator& a = Allocator()) const;
```

8 *Returns:* `this->address().to_string(a) + "/" + to_string(prefix_length()).c_str()`.

21.12.3 `ip::network_v6` comparisons [internet.network.v6.comparisons]

```
constexpr bool operator==(const network_v6& a, const network_v6& b) noexcept;
```

1 *Returns:* `true` if `a.address() == b.address()` and `a.prefix_length() == b.prefix_length()`, otherwise `false`.

```
constexpr bool operator!=(const network_v6& a, const network_v6& b) noexcept;
```

2 *Returns:* `!(a == b)`.

21.12.4 `ip::network_v6` creation [internet.network.v6.creation]

```
constexpr network_v6 make_network_v6(const address_v6& addr, int prefix_len);
```

1 *Returns:* `network_v6(addr, prefix_len)`.

```
network_v6 make_network_v6(const char* str);
network_v6 make_network_v6(const char* str, error_code& ec) noexcept;
network_v6 make_network_v6(const string& str);
network_v6 make_network_v6(const string& str, error_code& ec) noexcept;
network_v6 make_network_v6(string_view str);
network_v6 make_network_v6(string_view str, error_code& ec) noexcept;
```

2 *Returns:* If `str` contains a value of the form address `'/'` prefix-length, a `network_v6` object constructed with the result of applying `make_address_v6()` to the address portion of the string, and the result of converting prefix-length to an integer of type `int`. Otherwise returns `network_v6()` and sets `ec` to reflect the error.

3 *Error conditions:*

(3.1) — `errc::invalid_argument` — if `str` is not a valid textual representation of an IPv6 address and prefix length.

21.12.5 `ip::network_v6` I/O [internet.network.v6.io]

```
template<class CharT, class Traits>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os, const network_v6& net);
```

1 *Returns:* `os << net.to_string().c_str()`.

21.13 Class template `ip::basic_endpoint` [internet.endpoint]

1 An object of type `basic_endpoint<InternetProtocol>` represents a protocol-specific endpoint, where an endpoint consists of an IP address and port number. Endpoints may be used to identify sources and destinations for socket connections and datagrams.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  template<class InternetProtocol>
  class basic_endpoint
  {
  public:
    // types:
    using protocol_type = InternetProtocol;

    // constructors:
    constexpr basic_endpoint() noexcept;
    constexpr basic_endpoint(const protocol_type& proto,
                             port_type port_num) noexcept;
    constexpr basic_endpoint(const ip::address& addr,
                             port_type port_num) noexcept;

    // members:
    constexpr protocol_type protocol() const noexcept;
    constexpr ip::address address() const noexcept;
    void address(const ip::address& addr) noexcept;
    constexpr port_type port() const noexcept;
    void port(port_type port_num) noexcept;
  };

  // basic_endpoint comparisons:
  template<class InternetProtocol>
    constexpr bool operator==(const basic_endpoint<InternetProtocol>& a,
                              const basic_endpoint<InternetProtocol>& b) noexcept;
  template<class InternetProtocol>
    constexpr bool operator!=(const basic_endpoint<InternetProtocol>& a,
                              const basic_endpoint<InternetProtocol>& b) noexcept;
  template<class InternetProtocol>
    constexpr bool operator< (const basic_endpoint<InternetProtocol>& a,
                              const basic_endpoint<InternetProtocol>& b) noexcept;
  template<class InternetProtocol>
    constexpr bool operator> (const basic_endpoint<InternetProtocol>& a,
                              const basic_endpoint<InternetProtocol>& b) noexcept;
  template<class InternetProtocol>
    constexpr bool operator<=(const basic_endpoint<InternetProtocol>& a,
                              const basic_endpoint<InternetProtocol>& b) noexcept;
  template<class InternetProtocol>
    constexpr bool operator>=(const basic_endpoint<InternetProtocol>& a,
                              const basic_endpoint<InternetProtocol>& b) noexcept;

  // basic_endpoint I/O:
  template<class CharT, class Traits, class InternetProtocol>
    basic_ostream<CharT, Traits>& operator<<(
      basic_ostream<CharT, Traits>& os,
      const basic_endpoint<InternetProtocol>& ep);

} // namespace ip
```

```
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 Instances of the `basic_endpoint` class template meet the requirements for an `Endpoint` (18.2.4).

3 Extensible implementations provide the following member functions:

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  template<class InternetProtocol>
  class basic_endpoint
  {
  public:
    void* data() noexcept;
    const void* data() const noexcept;
    constexpr size_t size() const noexcept;
    void resize(size_t s);
    constexpr size_t capacity() const noexcept;
    // remainder unchanged
  private:
    union
    {
      sockaddr_in v4_;
      sockaddr_in6 v6_;
    } data_; // exposition only
  };

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

21.13.1 `ip::basic_endpoint` constructors [internet.endpoint.cons]

```
constexpr basic_endpoint() noexcept;
```

1 *Postconditions:* `this->address() == ip::address()` and `port() == 0`.

```
constexpr basic_endpoint(const protocol_type& proto,
                         port_type port_num) noexcept;
```

2 *Requires:* `proto == protocol_type::v4() || proto == protocol_type::v6()`.

3 *Postconditions:*

(3.1) — If `proto == protocol_type::v6()`, `this->address() == ip::address_v6()`; otherwise, `this->address() == ip::address_v4()`.

(3.2) — `port() == port_num`.

```
constexpr basic_endpoint(const ip::address& addr,
                         port_type port_num) noexcept;
```

4 *Postconditions:* `this->address() == addr` and `port() == port_num`.

21.13.2 `ip::basic_endpoint` members [internet.endpoint.members]

```
constexpr protocol_type protocol() const noexcept;
```

1 *Returns:* `protocol_type::v6()` if the expression `this->address().is_v6()` is `true`, otherwise `protocol_type::v4()`.

```
constexpr ip::address address() const noexcept;
```

2 *Returns:* The address associated with the endpoint.

```
void address(const ip::address& addr) noexcept;
```

3 *Postconditions:* `this->address() == addr`.

```
constexpr port_type port() const noexcept;
```

4 *Returns:* The port number associated with the endpoint.

```
void port(port_type port_num) noexcept;
```

5 *Postconditions:* `port() == port_num`.

21.13.3 `ip::basic_endpoint` comparisons [internet.endpoint.comparisons]

```
template<class InternetProtocol>
  constexpr bool operator==(const basic_endpoint<InternetProtocol>& a,
                            const basic_endpoint<InternetProtocol>& b) noexcept;
```

1 *Returns:* `a.address() == b.address() && a.port() == b.port())`.

```
template<class InternetProtocol>
  constexpr bool operator!=(const basic_endpoint<InternetProtocol>& a,
                            const basic_endpoint<InternetProtocol>& b) noexcept;
```

2 *Returns:* `!(a == b)`.

```
template<class InternetProtocol>
  constexpr bool operator< (const basic_endpoint<InternetProtocol>& a,
                            const basic_endpoint<InternetProtocol>& b) noexcept;
```

3 *Returns:* `a.address() < b.address() || (!(b.address() < a.address()) && a.port() < b.port())`.

```
template<class InternetProtocol>
  constexpr bool operator> (const basic_endpoint<InternetProtocol>& a,
                            const basic_endpoint<InternetProtocol>& b) noexcept;
```

4 *Returns:* `b < a`.

```
template<class InternetProtocol>
  constexpr bool operator<=(const basic_endpoint<InternetProtocol>& a,
                            const basic_endpoint<InternetProtocol>& b) noexcept;
```

5 *Returns:* `!(b < a)`.

```
template<class InternetProtocol>
  constexpr bool operator>=(const basic_endpoint<InternetProtocol>& a,
                            const basic_endpoint<InternetProtocol>& b) noexcept;
```

6 *Returns:* `!(a < b)`.

21.13.4 ip::basic_endpoint I/O [internet.endpoint.io]

```
template<class CharT, class Traits, class InternetProtocol>
  basic_ostream<CharT, Traits>& operator<<(
    basic_ostream<CharT, Traits>& os,
    const basic_endpoint<InternetProtocol>& ep);
```

1 *Effects:* Outputs a representation of the endpoint to the stream, as if it were implemented as follows:

```
basic_ostringstream<CharT, Traits> ss;
if (ep.protocol() == basic_endpoint<InternetProtocol>::protocol_type::v6())
  ss << "[" << ep.address() << "]";
else
  ss << ep.address();
ss << ":" << ep.port();
os << ss.str();
```

2 *Returns:* os.

3 [*Note:* The representation of the endpoint when it contains an IP version 6 address is based on [RFC2732]. — *end note*]

21.13.5 ip::basic_endpoint members (extensible implementations) [internet.endpoint.extensible]

```
void* data() noexcept;
```

1 *Returns:* std::addressof(data_).

```
const void* data() const noexcept;
```

2 *Returns:* std::addressof(data_).

```
constexpr size_t size() const noexcept;
```

3 *Returns:* sizeof(sockaddr_in6) if protocol().family() == AF_INET6, otherwise sizeof(sockaddr_in).

```
void resize(size_t s);
```

4 *Remarks:* length_error if the condition protocol().family() == AF_INET6 && s != sizeof(sockaddr_in6) || protocol().family() == AF_INET4 && s != sizeof(sockaddr_in) is true.

```
constexpr size_t capacity() const noexcept;
```

5 *Returns:* sizeof(data_).

21.14 Class template ip::basic_resolver_entry [internet.resolver.entry]

1 An object of type basic_resolver_entry<InternetProtocol> represents a single element in the results returned by a name resolution operation.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  template<class InternetProtocol>
  class basic_resolver_entry
```

```
{
public:
  // types:
  using protocol_type = InternetProtocol;
  using endpoint_type = typename InternetProtocol::endpoint;

  // constructors:
  basic_resolver_entry();
  basic_resolver_entry(const endpoint_type& ep,
                       string_view h,
                       string_view s);

  // members:
  endpoint_type endpoint() const;
  operator endpoint_type() const;
  template<class Allocator = allocator<char>>
    basic_string<char, char_traits<char>, Allocator>
      host_name(const Allocator& a = Allocator()) const;
  template<class Allocator = allocator<char>>
    basic_string<char, char_traits<char>, Allocator>
      service_name(const Allocator& a = Allocator()) const;
};

// basic_resolver_entry comparisons:
template<class InternetProtocol>
  bool operator==(const basic_resolver_entry<InternetProtocol>& a,
                  const basic_resolver_entry<InternetProtocol>& b);
template<class InternetProtocol>
  bool operator!=(const basic_resolver_entry<InternetProtocol>& a,
                  const basic_resolver_entry<InternetProtocol>& b);

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

21.14.1 ip::basic_resolver_entry constructors [internet.resolver.entry.cons]

`basic_resolver_entry();`

1 *Effects:* Equivalent to `basic_resolver_entry<InternetProtocol>(endpoint_type(), "", "")`.

```
basic_resolver_entry(const endpoint_type& ep,
                     string_view h,
                     string_view s);
```

2 *Postconditions:*

(2.1) — `endpoint() == ep`.

(2.2) — `host_name() == h`.

(2.3) — `service_name() == s`.

21.14.2 ip::basic_resolver_entry members [internet.resolver.entry.members]

`endpoint_type endpoint() const;`

1 *Returns:* The endpoint associated with the resolver entry.

```
operator endpoint_type() const;
```

2 *Returns:* `endpoint()`.

```
template<class Allocator = allocator<char>>
  basic_string<char, char_traits<char>, Allocator>
    host_name(const Allocator& a = Allocator()) const;
```

3 *Returns:* The host name associated with the resolver entry.

4 *Remarks:* Ill-formed unless `allocator_traits<Allocator>::value_type` is `char`.

```
template<class Allocator = allocator<char>>
  basic_string<char, char_traits<char>, Allocator>
    service_name(const Allocator& a = Allocator()) const;
```

5 *Returns:* The service name associated with the resolver entry.

6 *Remarks:* Ill-formed unless `allocator_traits<Allocator>::value_type` is `char`.

21.14.3 `op::basic_resolver_entry` comparisons [internet.resolver.entry.comparisons]

```
template<class InternetProtocol>
  bool operator==(const basic_resolver_entry<InternetProtocol>& a,
                  const basic_resolver_entry<InternetProtocol>& b);
```

1 *Returns:* `a.endpoint() == b.endpoint() && a.host_name() == b.host_name() && a.service_name() == b.service_name()`.

```
template<class InternetProtocol>
  bool operator!=(const basic_resolver_entry<InternetProtocol>& a,
                  const basic_resolver_entry<InternetProtocol>& b);
```

2 *Returns:* `!(a == b)`.

21.15 Class template `ip::basic_resolver_results` [internet.resolver.results]

1 An object of type `basic_resolver_results<InternetProtocol>` represents a sequence of `basic_resolver_entry<InternetProtocol>` elements resulting from a single name resolution operation.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  template<class InternetProtocol>
  class basic_resolver_results
  {
  public:
    // types:
    using protocol_type = InternetProtocol;
    using endpoint_type = typename protocol_type::endpoint;
    using value_type = basic_resolver_entry<protocol_type>;
    using const_reference = const value_type&;
    using reference = value_type&;
    using const_iterator = implementation-defined;
    using iterator = const_iterator;
```

```
    using difference_type = ptrdiff_t;
    using size_type = size_t;

    // construct / copy / destroy:
    basic_resolver_results();
    basic_resolver_results(const basic_resolver_results& rhs);
    basic_resolver_results(basic_resolver_results&& rhs) noexcept;
    basic_resolver_results& operator=(const basic_resolver_results& rhs);
    basic_resolver_results& operator=(basic_resolver_results&& rhs);
    ~basic_resolver_results();

    // size:
    size_type size() const noexcept;
    size_type max_size() const noexcept;
    bool empty() const noexcept;

    // element access:
    const_iterator begin() const;
    const_iterator end() const;
    const_iterator cbegin() const;
    const_iterator cend() const;

    // swap:
    void swap(basic_resolver_results& that) noexcept;
  };

  // basic_resolver_results comparisons:
  template<class InternetProtocol>
    bool operator==(const basic_resolver_results<InternetProtocol>& a,
                    const basic_resolver_results<InternetProtocol>& b);
  template<class InternetProtocol>
    bool operator!=(const basic_resolver_results<InternetProtocol>& a,
                    const basic_resolver_results<InternetProtocol>& b);

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The class template `basic_resolver_results` satisfies the requirements of a sequence container (C++Std [sequence.reqmts]), except that only the operations defined for const-qualified sequence containers are supported. The class template `basic_resolver_results` supports forward iterators.

3 A default-constructed `basic_resolver_results` object is empty. A non-empty results object is obtained only by calling a `basic_resolver` object's `wait` or `async_wait` operations, or otherwise by copy construction, move construction, assignment, or swap from another non-empty results object.

21.15.1 `ip::basic_resolver_results` constructors [internet.resolver.results.cons]

`basic_resolver_results();`

1 *Postconditions:* `size() == 0`.

`basic_resolver_results(const basic_resolver_results& rhs);`

2 *Postconditions:* `*this == rhs`.

```
basic_resolver_results(basic_resolver_results&& rhs) noexcept;
```

3 *Postconditions:* `*this` is equal to the prior value of `rhs`.

21.15.2 ip::basic_resolver_results assignment [internet.resolver.results.assign]

```
basic_resolver_results& operator=(const basic_resolver_results& rhs);
```

1 *Postconditions:* `*this == rhs`.

2 *Returns:* `*this`.

```
basic_resolver_results& operator=(basic_resolver_results& rhs) noexcept;
```

3 *Postconditions:* `*this` is equal to the prior value of `rhs`.

4 *Returns:* `*this`.

21.15.3 ip::basic_resolver_results size [internet.resolver.results.size]

```
size_type size() const noexcept;
```

1 *Returns:* The number of `basic_resolver_entry` elements in `*this`.

```
size_type max_size() const noexcept;
```

2 *Returns:* The maximum number of `basic_resolver_entry` elements that can be stored in `*this`.

```
bool empty() const noexcept;
```

3 *Returns:* `size() == 0`.

21.15.4 ip::basic_resolver_results element access [internet.resolver.results.access]

```
const_iterator begin() const;
const_iterator cbegin() const;
```

1 *Returns:* A starting iterator that enumerates over all the `basic_resolver_entry` elements stored in `*this`.

```
const_iterator end() const;
const_iterator cend() const;
```

2 *Returns:* A terminating iterator that enumerates over all the `basic_resolver_entry` elements stored in `*this`.

21.15.5 ip::basic_resolver_results swap [internet.resolver.results.swap]

```
void swap(basic_resolver_results& that) noexcept;
```

1 *Postconditions:* `*this` is equal to the prior value of `that`, and `that` is equal to the prior value of `*this`.

21.15.6 ip::basic_resolver_results comparisons [internet.resolver.results.comparisons]

```
template<class InternetProtocol>
  bool operator==(const basic_resolver_results<InternetProtocol>& a,
                  const basic_resolver_results<InternetProtocol>& b);
```

1 *Returns:* `a.size() == b.size() && equal(a.cbegin(), a.cend(), b.cbegin())`.

```
template<class InternetProtocol>
  bool operator!=(const basic_resolver_results<InternetProtocol>& a,
                  const basic_resolver_results<InternetProtocol>& b);
```

2 *Returns:* `!(a == b)`.

21.16 Class `ip::resolver_base` [internet.resolver.base]

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  class resolver_base
  {
  public:
    using flags = T1;
    static const flags passive;
    static const flags canonical_name;
    static const flags numeric_host;
    static const flags numeric_service;
    static const flags v4_mapped;
    static const flags all_matching;
    static const flags address_configured;

  protected:
    resolver_base();
    ~resolver_base();
  };

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

1 `resolver_base` defines a bitmask type, `flags`, with the bitmask elements shown in Table 37.

Table 37 — Resolver flags

Constant name	POSIX macro	Definition or notes
`passive`	`AI_PASSIVE`	Returned endpoints are intended for use as locally bound socket endpoints.
`canonical_name`	`AI_CANONNAME`	Determine the canonical name of the host specified in the query.
`numeric_host`	`AI_NUMERICHOST`	Host name should be treated as a numeric string defining an IPv4 or IPv6 address and no host name resolution should be attempted.
`numeric_service`	`AI_NUMERICSERV`	Service name should be treated as a numeric string defining a port number and no service name resolution should be attempted.
`v4_mapped`	`AI_V4MAPPED`	If the protocol is specified as an IPv6 protocol, return IPv4-mapped IPv6 addresses on finding no IPv6 addresses.

Table 37 — Resolver flags (continued)

Constant name	POSIX macro	Definition or notes
`all_matching`	`AI_ALL`	If used with `v4_mapped`, return all matching IPv6 and IPv4 addresses.
`address_configured`	`AI_ADDRCONFIG`	Only return IPv4 addresses if a non-loopback IPv4 address is configured for the system. Only return IPv6 addresses if a non-loopback IPv6 address is configured for the system.

21.17 Class template `ip::basic_resolver` [internet.resolver]

1 Objects of type `basic_resolver<InternetProtocol>` are used to perform name resolution. Name resolution is the translation of a host name and service name into a sequence of endpoints, or the translation of an endpoint into its corresponding host name and service name.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  template<class InternetProtocol>
  class basic_resolver : public resolver_base
  {
  public:
    // types:

    using executor_type = io_context::executor_type;
    using protocol_type = InternetProtocol;
    using endpoint_type = typename InternetProtocol::endpoint;
    using results_type = basic_resolver_results<InternetProtocol>;

    // construct / copy / destroy:

    explicit basic_resolver(io_context& ctx);
    basic_resolver(const basic_resolver&) = delete;
    basic_resolver(basic_resolver&& rhs) noexcept;

    ~basic_resolver();

    basic_resolver& operator=(const basic_resolver&) = delete;
    basic_resolver& operator=(basic_resolver&& rhs);

    // basic_resolver operations:

    executor_type get_executor() noexcept;

    void cancel();

    results_type resolve(string_view host_name, string_view service_name);
    results_type resolve(string_view host_name, string_view service_name,
                         error_code& ec);
    results_type resolve(string_view host_name, string_view service_name,
                         flags f);
```

```
    results_type resolve(string_view host_name, string_view service_name,
                         flags f, error_code& ec);

    template<class CompletionToken>
      DEDUCED async_resolve(string_view host_name, string_view service_name,
                            CompletionToken&& token);
    template<class CompletionToken>
      DEDUCED async_resolve(string_view host_name, string_view service_name,
                            flags f, CompletionToken&& token);

    results_type resolve(const protocol_type& protocol,
                         string_view host_name, string_view service_name);
    results_type resolve(const protocol_type& protocol,
                         string_view host_name, string_view service_name,
                         error_code& ec);
    results_type resolve(const protocol_type& protocol,
                         string_view host_name, string_view service_name,
                         flags f);
    results_type resolve(const protocol_type& protocol,
                         string_view host_name, string_view service_name,
                         flags f, error_code& ec);

    template<class CompletionToken>
      DEDUCED async_resolve(const protocol_type& protocol,
                            string_view host_name, string_view service_name,
                            CompletionToken&& token);
    template<class CompletionToken>
      DEDUCED async_resolve(const protocol_type& protocol,
                            string_view host_name, string_view service_name,
                            flags f, CompletionToken&& token);

    results_type resolve(const endpoint_type& e);
    results_type resolve(const endpoint_type& e, error_code& ec);

    template<class CompletionToken>
      DEDUCED async_resolve(const endpoint_type& e,
                            CompletionToken&& token);
  };

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

21.17.1 ip::basic_resolver constructors [internet.resolver.cons]

`explicit basic_resolver(io_context& ctx);`

1 *Postconditions:* `get_executor() == ctx.get_executor()`.

`basic_resolver(basic_resolver&& rhs) noexcept;`

2 *Effects:* Move constructs an object of class `basic_resolver<InternetProtocol>` that refers to the state originally represented by `rhs`.

3 *Postconditions:* `get_executor() == rhs.get_executor()`.

21.17.2 ip::basic_resolver destructor [internet.resolver.dtor]

```
~basic_resolver();
```

1 *Effects:* Destroys the resolver, canceling all asynchronous operations associated with this resolver as if by calling `cancel()`.

21.17.3 ip::basic_resolver assignment [internet.resolver.assign]

```
basic_resolver& operator=(basic_resolver&& rhs);
```

1 *Effects:* Cancels all outstanding asynchronous operations associated with `*this` as if by calling `cancel()`, then moves into `*this` the state originally represented by `rhs`.

2 *Postconditions:* `get_executor() == ctx.get_executor()`.

3 *Returns:* `*this`.

21.17.4 ip::basic_resolver operations [internet.resolver.ops]

```
executor_type get_executor() noexcept;
```

1 *Returns:* The associated executor.

```
void cancel();
```

2 *Effects:* Cancels all outstanding asynchronous resolve operations associated with `*this`. Completion handlers for canceled operations are passed an error code `ec` such that `ec == errc::operation_canceled` yields `true`.

3 *Remarks:* Does not block (C++Std [defns.block]) the calling thread pending completion of the canceled operations.

```
results_type resolve(string_view host_name, string_view service_name);
results_type resolve(string_view host_name, string_view service_name,
                     error_code& ec);
```

4 *Returns:* `resolve(host_name, service_name, resolver_base::flags(), ec)`.

```
results_type resolve(string_view host_name, string_view service_name,
                     flags f);
results_type resolve(string_view host_name, string_view service_name,
                     flags f, error_code& ec);
```

5 *Effects:* If `host_name.data() != nullptr`, let H be an NTBS constructed from `host_name`; otherwise, let H be `nullptr`. If `service_name.data() != nullptr`, let S be an NTBS constructed from `service_name`; otherwise, let S be `nullptr`. Resolves a host name and service name, as if by POSIX:

```
addrinfo hints;
hints.ai_flags = static_cast<int>(f);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = endpoint_type().protocol().type();
hints.ai_protocol = endpoint_type().protocol().protocol();
hints.ai_addr = nullptr;
hints.ai_addrlen = 0;
hints.ai_canonname = nullptr;
hints.ai_next = nullptr;
addrinfo* result = nullptr;
getaddrinfo(H, S, &hints, &result);
```

6 *Returns:* On success, a non-empty results object containing the results of the resolve operation. Otherwise `results_type()`.

```
template<class CompletionToken>
  DEDUCED async_resolve(string_view host_name, string_view service_name,
                        CompletionToken&& token);
```

7 *Returns:* `async_resolve(host_name, service_name, resolver_base::flags(), forward<CompletionToken>(token))`.

```
template<class CompletionToken>
  DEDUCED async_resolve(string_view host_name, string_view service_name,
                        flags f, CompletionToken&& token);
```

8 *Completion signature:* `void(error_code ec, results_type r)`.

9 *Effects:* If `host_name.data() != nullptr`, let H be an NTBS constructed from `host_name`; otherwise, let H be `nullptr`. If `service_name.data() != nullptr`, let S be an NTBS constructed from `service_name`; otherwise, let S be `nullptr`. Initiates an asynchronous operation to resolve a host name and service name, as if by POSIX:

```
addrinfo hints;
hints.ai_flags = static_cast<int>(f);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = endpoint_type().protocol().type();
hints.ai_protocol = endpoint_type().protocol().protocol();
hints.ai_addr = nullptr;
hints.ai_addrlen = 0;
hints.ai_canonname = nullptr;
hints.ai_next = nullptr;
addrinfo* result = nullptr;
getaddrinfo(H, S, &hints, &result);
```

On success, `r` is a non-empty results object containing the results of the resolve operation. Otherwise, `r` is `results_type()`.

```
results_type resolve(const protocol_type& protocol,
                     string_view host_name, string_view service_name);
results_type resolve(const protocol_type& protocol,
                     string_view host_name, string_view service_name,
                     error_code& ec);
```

10 *Returns:* `resolve(protocol, host_name, service_name, resolver_base::flags(), ec)`.

```
results_type resolve(const protocol_type& protocol,
                     string_view host_name, string_view service_name,
                     flags f);
results_type resolve(const protocol_type& protocol,
                     string_view host_name, string_view service_name,
                     flags f, error_code& ec);
```

11 *Effects:* If `host_name.data() != nullptr`, let H be an NTBS constructed from `host_name`; otherwise, let H be `nullptr`. If `service_name.data() != nullptr`, let S be an NTBS constructed from `service_name`; otherwise, let S be `nullptr`. Resolves a host name and service name, as if by POSIX:

```
addrinfo hints;
hints.ai_flags = static_cast<int>(f);
hints.ai_family = protocol.family();
```

```
hints.ai_socktype = protocol.type();
hints.ai_protocol = protocol.protocol();
hints.ai_addr = nullptr;
hints.ai_addrlen = 0;
hints.ai_canonname = nullptr;
hints.ai_next = nullptr;
addrinfo* result = nullptr;
getaddrinfo(H, S, &hints, &result);
```

12 *Returns:* On success, a non-empty results object containing the results of the resolve operation. Otherwise `results_type()`.

```
template<class CompletionToken>
  DEDUCED async_resolve(const protocol_type& protocol,
                        string_view host_name, string_view service_name,
                        CompletionToken&& token);
```

13 *Returns:* `async_resolve(protocol, host_name, service_name, resolver_base::flags(), forward<CompletionToken>(token))`.

```
template<class CompletionToken>
  DEDUCED async_resolve(const protocol& protocol,
                        string_view host_name, string_view service_name,
                        flags f, CompletionToken&& token);
```

14 *Completion signature:* `void(error_code ec, results_type r)`.

15 *Effects:* If `host_name.data() != nullptr`, let H be an NTBS constructed from `host_name`; otherwise, let H be `nullptr`. If `service_name.data() != nullptr`, let S be an NTBS constructed from `service_name`; otherwise, let S be `nullptr`. Initiates an asynchronous operation to resolve a host name and service name, as if by POSIX:

```
addrinfo hints;
hints.ai_flags = static_cast<int>(f);
hints.ai_family = protocol.family();
hints.ai_socktype = protocol.type();
hints.ai_protocol = protocol.protocol();
hints.ai_addr = nullptr;
hints.ai_addrlen = 0;
hints.ai_canonname = nullptr;
hints.ai_next = nullptr;
addrinfo* result = nullptr;
getaddrinfo(H, S, &hints, &result);
```

On success, `r` is a non-empty results object containing the results of the resolve operation. Otherwise, `r` is `results_type()`.

```
results_type resolve(const endpoint_type& e);
results_type resolve(const endpoint_type& e, error_code& ec);
```

16 *Effects:* Let `S1` and `S2` be implementation-defined values that are sufficiently large to hold the host name and service name respectively. Resolves an endpoint as if by POSIX:

```
char host_name[S1];
char service_name[S2];
int flags = 0;
if (endpoint_type().protocol().type() == SOCK_DGRAM)
  flags |= NI_DGRAM;
```

```
int result = getnameinfo((const sockaddr*)e.data(), e.size(),
                         host_name, S1,
                         service_name, S2,
                         flags);
if (result != 0)
{
  flags |= NI_NUMERICSERV;
  result = getnameinfo((const sockaddr*)e.data(), e.size(),
                       host_name, S1,
                       service_name, S2,
                       flags);
}
```

17 *Returns:* On success, a results object with `size() == 1` containing the results of the resolve operation. Otherwise `results_type()`.

```
template<class CompletionToken>
  DEDUCED async_resolve(const endpoint_type& e,
                        CompletionToken&& token);
```

18 *Completion signature:* `void(error_code ec, results_type r)`.

19 *Effects:* Let `S1` and `S2` be implementation-defined values that are sufficiently large to hold the host name and service name respectively. Initiates an asynchronous operation to resolve an endpoint as if by POSIX:

```
char host_name[S1];
char service_name[S2];
int flags = 0;
if (endpoint_type().protocol().type() == SOCK_DGRAM)
  flags |= NI_DGRAM;
int result = getnameinfo((const sockaddr*)e.data(), e.size(),
                         host_name, S1,
                         service_name, S2,
                         flags);
if (result != 0)
{
  flags |= NI_NUMERICSERV;
  result = getnameinfo((const sockaddr*)e.data(), e.size(),
                       host_name, S1,
                       service_name, S2,
                       flags);
}
```

On success, `r` is a results object with `size() == 1` containing the results of the resolve operation; otherwise, `r` is `results_type()`.

21.18 Host name functions [internet.host.name]

```
string host_name();
string host_name(error_code& ec);
template<class Allocator>
  basic_string<char, char_traits<char>, Allocator>
    host_name(const Allocator& a);
template<class Allocator>
  basic_string<char, char_traits<char>, Allocator>
    host_name(const Allocator& a, error_code& ec);
```

1 *Returns:* The standard host name for the current machine, determined as if by POSIX `gethostname`.

2 *Remarks:* In the last two overloads, ill-formed unless `allocator_traits<Allocator>::value_type` is `char`.

21.19 Class `ip::tcp` [internet.tcp]

1 The class `tcp` encapsulates the types and flags necessary for TCP sockets.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  class tcp
  {
  public:
    // types:
    using endpoint = basic_endpoint<tcp>;
    using resolver = basic_resolver<tcp>;
    using socket = basic_stream_socket<tcp>;
    using acceptor = basic_socket_acceptor<tcp>;
    using iostream = basic_socket_iostream<tcp>;
    class no_delay;

    // static members:
    static constexpr tcp v4() noexcept;
    static constexpr tcp v6() noexcept;

    tcp() = delete;
  };

  // tcp comparisons:
  constexpr bool operator==(const tcp& a, const tcp& b) noexcept;
  constexpr bool operator!=(const tcp& a, const tcp& b) noexcept;

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The `tcp` class meets the requirements for an `InternetProtocol` (21.2.1).

3 Extensible implementations provide the following member functions:

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  class tcp
  {
  public:
    constexpr int family() const noexcept;
    constexpr int type() const noexcept;
```

```
    constexpr int protocol() const noexcept;
    // remainder unchanged
  };

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

4 The return values for these member functions are listed in Table 38.

Table 38 — Behavior of extensible `ip::tcp` implementations

value	family()	type()	protocol()
`tcp::v4()`	`AF_INET`	`SOCK_STREAM`	`IPPROTO_TCP`
`tcp::v6()`	`AF_INET6`	`SOCK_STREAM`	`IPPROTO_TCP`

5 [*Note:* The constants `AF_INET`, `AF_INET6` and `SOCK_STREAM` are defined in the POSIX header file `<sys/socket.h>`. The constant `IPPROTO_TCP` is defined in the POSIX header file `<netinet/in.h>`. — *end note*]

21.19.1 `ip::tcp` comparisons [internet.tcp.comparisons]

```
constexpr bool operator==(const tcp& a, const tcp& b) noexcept;
```

1 *Returns:* A boolean indicating whether two objects of class `tcp` are equal, such that the expression `tcp::v4() == tcp::v4()` is `true`, the expression `tcp::v6() == tcp::v6()` is `true`, and the expression `tcp::v4() == tcp::v6()` is `false`.

```
constexpr bool operator!=(const tcp& a, const tcp& b) noexcept;
```

2 *Returns:* `!(a == b)`.

21.20 Class `ip::udp` [internet.udp]

1 The class `udp` encapsulates the types and flags necessary for UDP sockets.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  class udp
  {
  public:
    // types:
    using endpoint = basic_endpoint<udp>;
    using resolver = basic_resolver<udp>;
    using socket = basic_datagram_socket<udp>;

    // static members:
    static constexpr udp v4() noexcept;
    static constexpr udp v6() noexcept;

    udp() = delete;
```

```
  };

  // udp comparisons:
  constexpr bool operator==(const udp& a, const udp& b) noexcept;
  constexpr bool operator!=(const udp& a, const udp& b) noexcept;

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 The `udp` class meets the requirements for an `InternetProtocol` (21.2.1).

3 Extensible implementations provide the following member functions:

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {

  class udp
  {
  public:
    constexpr int family() const noexcept;
    constexpr int type() const noexcept;
    constexpr int protocol() const noexcept;
    // remainder unchanged
  };

} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

4 The return values for these member functions are listed in Table 39.

Table 39 — Behavior of extensible `ip::udp` implementations

value	family()	type()	protocol()
`udp::v4()`	`AF_INET`	`SOCK_DGRAM`	`IPPROTO_UDP`
`udp::v6()`	`AF_INET6`	`SOCK_DGRAM`	`IPPROTO_UDP`

5 [*Note:* The constants `AF_INET`, `AF_INET6` and `SOCK_DGRAM` are defined in the POSIX header file `<sys/socket.h>`. The constant `IPPROTO_UDP` is defined in the POSIX header file `<netinet/in.h>`. — *end note*]

21.20.1 `ip::udp` comparisons [internet.udp.comparisons]

```
constexpr bool operator==(const udp& a, const udp& b) noexcept;
```

1 *Returns:* A boolean indicating whether two objects of class `udp` are equal, such that the expression `udp::v4() == udp::v4()` is `true`, the expression `udp::v6() == udp::v6()` is `true`, and the expression `udp::v4() == udp::v6()` is `false`.

```
constexpr bool operator!=(const udp& a, const udp& b) noexcept;
```

2 *Returns:* `!(a == b)`.

21.21 Internet socket options [internet.socket.opt]

1 In Table 40, let *C* denote a socket option class; let *L* identify the POSIX macro to be passed as the `level` argument to POSIX `setsockopt` and getsockopt; let *N* identify the POSIX macro to be passed as the `option_name` argument to POSIX `setsockopt` and `getsockopt`; let *T* identify the type of the value whose address will be passed as the `option_value` argument to POSIX `setsockopt` and `getsockopt`; let `p` denote a (possibly const) value of a type meeting the protocol (18.2.6) requirements, as passed to the socket option's `level` and `name` member functions; and let `F` be the value of `p.family()`.

Table 40 — Internet socket options

C	***L***	***N***	***T***	**Requirements, definition or notes**
`ip::tcp::no_delay`	`IPPROTO_TCP`	`TCP_NODELAY`	`int`	Satisfies the `BooleanSocketOption` (18.2.10) type requirements. Determines whether a TCP socket will avoid coalescing of small segments. [*Note:* That is, setting this option disables the Nagle algorithm. — *end note*]

Table 40 — Internet socket options (continued)

C	*L*	*N*	*T*	**Requirements, definition or notes**
`ip::v6_only`	`IPPROTO_IPV6`	`IPV6_V6ONLY`	`int`	Satisfies the `BooleanSocketOption` (18.2.10) type requirements. Determines whether a socket created for an IPv6 protocol is restricted to IPv6 communications only. Implementations are not required to support setting the `v6_only` option to `false`, and the initial value of the `v6_only` option for a socket is implementation-defined. [*Note:* As not all operating systems support dual stack IP networking. Some operating systems that do provide dual stack support offer a configuration option to disable it or to set the initial value of the `v6_only` socket option. — *end note*]
`ip::unicast::hops`	`IPPROTO_IPV6` if `F == AF_INET6`, otherwise `IPPROTO_IP`	`IPV6_UNICAST_HOPS` if `F == AF_INET6`, otherwise `IP_TTL`	`int`	Satisfies the `IntegerSocketOption` (18.2.11) type requirements. Specifies the default number of hops (also known as time-to-live or TTL) on outbound datagrams. The constructor and assignment operator for the `ip::unicast::hops` class throw `out_of_range` if the `int` argument is not in the range `[0, 255]`.

Table 40 — Internet socket options (continued)

C	*L*	*N*	*T*	**Requirements, definition or notes**
`ip::multicast::join_group`	`IPPROTO_IPV6` if `F == AF_INET6`, otherwise `IPPROTO_IP`	`IPV6_JOIN_GROUP` if `F == AF_INET6`, otherwise `IP_ADD_MEMBERSHIP`	`ipv6_mreq` if `F == AF_INET6`, otherwise `ip_mreq`	Satisfies the `MulticastGroupSocketOption` (21.2.2) type requirements. Requests that the socket join the specified multicast group.
`ip::multicast::leave_group`	`IPPROTO_IPV6` if `F == AF_INET6`, otherwise `IPPROTO_IP`	`IPV6_LEAVE_GROUP` if `F == AF_INET6`, otherwise `IP_DROP_MEMBERSHIP`	`ipv6_mreq` if `F == AF_INET6`, otherwise `ip_mreq`	Satisfies the `MulticastGroupSocketOption` (21.2.2) type requirements. Requests that the socket leave the specified multicast group.
`ip::multicast::outbound_interface` (21.21.1)	`IPPROTO_IPV6` if `F == AF_INET6`, otherwise `IPPROTO_IP`	`IPV6_MULTICAST_IF` if `F == AF_INET6`, otherwise `IP_MULTICAST_IF`	`unsigned int` if `F == AF_INET6`, otherwise `in_addr`	Specifies the network interface to use for outgoing multicast datagrams.
`ip::multicast::hops`	`IPPROTO_IPV6` if `F == AF_INET6`, otherwise `IPPROTO_IP`	`IPV6_MULTICAST_HOPS` if `F == AF_INET6`, otherwise `IP_MULTICAST_TTL`	`int`	Satisfies the `IntegerSocketOption` (18.2.11) type requirements. Specifies the default number of hops (also known as time-to-live or TTL) on outbound datagrams. The constructor and assignment operator for the `ip::multicast::hops` class throw `out_of_range` if the `int` argument is not in the range `[0, 255]`.
`ip::multicast::enable_loopback`	`IPPROTO_IPV6` if `F == AF_INET6`, otherwise `IPPROTO_IP`	`IPV6_MULTICAST_LOOP` if `F == AF_INET6`, otherwise `IP_MULTICAST_LOOP`	`int`	Satisfies the `BooleanSocketOption` (18.2.10) type requirements. Determines whether multicast datagrams are delivered back to the local application.

21.21.1 Class `ip::multicast::outbound_interface` [internet.multicast.outbound]

1 The `outbound_interface` class represents a socket option that specifies the network interface to use for outgoing multicast datagrams.

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {
namespace multicast {

  class outbound_interface
  {
  public:
    // constructors:
    explicit outbound_interface(const address_v4& network_interface) noexcept;
    explicit outbound_interface(unsigned int network_interface) noexcept;
  };

} // namespace multicast
} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
} // namespace std
```

2 `outbound_interface` satisfies the requirements for `Destructible` (C++Std [destructible]), `CopyConstructible` (C++Std [copyconstructible]), `CopyAssignable` (C++Std [copyassignable]), and `SettableSocketOption` (18.2.9).

3 Extensible implementations provide the following member functions:

```
namespace std {
namespace experimental {
namespace net {
inline namespace v1 {
namespace ip {
namespace multicast {

  class outbound_interface
  {
  public:
    template<class Protocol> int level(const Protocol& p) const noexcept;
    template<class Protocol> int name(const Protocol& p) const noexcept;
    template<class Protocol> const void* data(const Protocol& p) const noexcept;
    template<class Protocol> size_t size(const Protocol& p) const noexcept;
    // remainder unchanged
  private:
      in_addr v4_value_; // exposition only
      unsigned int v6_value_; // exposition only
  };

} // namespace multicast
} // namespace ip
} // inline namespace v1
} // namespace net
} // namespace experimental
```

```
} // namespace std
```

```
explicit outbound_interface(const address_v4& network_interface) noexcept;
```

4 *Effects:* For extensible implementations, `v4_value_` is initialized to correspond to the IPv4 address `network_interface`, and `v6_value_` is zero-initialized.

```
explicit outbound_interface(unsigned int network_interface) noexcept;
```

5 *Effects:* For extensible implementations, `v6_value_` is initialized to `network_interface`, and `v4_value_` is zero-initialized.

```
template<class Protocol> int level(const Protocol& p) const noexcept;
```

6 *Returns:* `IPPROTO_IPV6` if `p.family() == AF_INET6`, otherwise `IPPROTO_IP`.

```
template<class Protocol> int name(const Protocol& p) const noexcept;
```

7 *Returns:* `IPV6_MULTICAST_IF` if `p.family() == AF_INET6`, otherwise `IP_MULTICAST_IF`.

```
template<class Protocol> const void* data(const Protocol& p) const noexcept;
```

8 *Returns:* `std::addressof(v6_value_)` if `p.family() == AF_INET6`, otherwise `std::addressof(v4_value_)`.

```
template<class Protocol> size_t size(const Protocol& p) const noexcept;
```

9 *Returns:* `sizeof(v6_value_)` if `p.family() == AF_INET6`, otherwise `sizeof(v4_value_)`.

Index

Index of library names

Index of implementation-defined behavior

The entries in this section are rough descriptions; exact specifications are at the indicated page in the general text.