Document Number: P1458R1
Date: 2019-02-20
Reply to: Marshall Clow
CppAlliance
mclow.lists@gmail.com

Mandating the Standard Library: Clause 16 - Language support library

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

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

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

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

— Drive-by fixes (hopefully very few)

This paper covers Clause 16 (Language Support)

As a drive-by fix, I have removed a bunch of empty descriptions of the form: "Effects: Constructs an object of class `Foo`."

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

— support.types.byteops 16.2.5
— new.delete.single 16.6.2.1
— new.delete.array 16.6.2.2
— new.delete.placement 16.6.2.3
— bad.alloc 16.6.3.1
— new.badlength 16.6.3.2
— ptr.launder 16.6.4
— bad.cast 16.7.3
— bad.typeid 16.7.4
— exception 16.9.2
— bad.exception 16.9.3
— propagation 16.9.6
— except.nested 16.9.7
— support.initlist.cons 16.10.2

Changes from R0:

— Reworked some of the "Expects" elements to use "is/was" instead of "shall".
— Added a "Mandates" to ptr.launder 16.6.4

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

```
git clone git@github.com:mclow/mandate.git
cd mandate
git diff master..chapter16 support.tex
```

16 Language support library [language.support]

16.1 General [support.general]

1 This Clause describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. It also describes the headers that declare these function signatures and define any related types.

2 The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for contract violation handling, support for exception processing, support for initializer lists, and other runtime support, as summarized in Table 34.

Table 34 — Language support library summary

	Subclause	Header(s)
16.2	Common definitions	`<cstddef>` `<cstdlib>`

16.3	Implementation properties	`<limits>`
		`<climits>`
		`<cfloat>`
		`<version>`

16.4	Integer types	`<cstdint>`
16.5	Start and termination	`<cstdlib>`
16.6	Dynamic memory management	`<new>`
16.7	Type identification	`<typeinfo>`
16.8	Contract violation handling	`<contract>`
16.9	Exception handling	`<exception>`
16.10	Initializer lists	`<initializer_list>`
16.11	Comparisons	`<compare>`

16.12	Other runtime support	`<csignal>`
		`<csetjmp>`
		`<cstdarg>`
		`<cstdlib>`

16.2 Common definitions [support.types]

16.2.1 Header <cstddef> synopsis [cstddef.syn]

```
namespace std {
  using ptrdiff_t = see below;
  using size_t = see below;
  using max_align_t = see below;
  using nullptr_t = decltype(nullptr);

  enum class byte : unsigned char {};

  // 16.2.5, byte type operations
  template<class IntType>
    constexpr byte& operator<<=(byte& b, IntType shift) noexcept;
  template<class IntType>
    constexpr byte operator<<(byte b, IntType shift) noexcept;
  template<class IntType>
    constexpr byte& operator>>=(byte& b, IntType shift) noexcept;
  template<class IntType>
    constexpr byte operator>>(byte b, IntType shift) noexcept;
```

```
  constexpr byte& operator|=(byte& l, byte r) noexcept;
  constexpr byte operator|(byte l, byte r) noexcept;
  constexpr byte& operator&=(byte& l, byte r) noexcept;
  constexpr byte operator&(byte l, byte r) noexcept;
  constexpr byte& operator^=(byte& l, byte r) noexcept;
  constexpr byte operator^(byte l, byte r) noexcept;
  constexpr byte operator~(byte b) noexcept;
  template<class IntType>
    constexpr IntType to_integer(byte b) noexcept;
}

#define NULL see below
#define offsetof(P, D) see below
```

1 The contents and meaning of the header `<cstddef>` are the same as the C standard library header `<stddef.h>`, except that it does not declare the type `wchar_t`, that it also declares the type `byte` and its associated operations (16.2.5), and as noted in 16.2.3 and 16.2.4.

See also: ISO C 7.19

16.2.2 Header <cstdlib> synopsis [cstdlib.syn]

```
namespace std {
  using size_t = see below;
  using div_t = see below;
  using ldiv_t = see below;
  using lldiv_t = see below;
}

#define NULL see below
#define EXIT_FAILURE see below
#define EXIT_SUCCESS see below
#define RAND_MAX see below
#define MB_CUR_MAX see below

namespace std {
  // Exposition-only function type aliases
  extern "C" using c-atexit-handler = void();                          // exposition only
  extern "C++" using atexit-handler = void();                          // exposition only
  extern "C" using c-compare-pred = int(const void*, const void*);     // exposition only
  extern "C++" using compare-pred = int(const void*, const void*);     // exposition only

  // 16.5, start and termination
  [[noreturn]] void abort() noexcept;
  int atexit(c-atexit-handler* func) noexcept;
  int atexit(atexit-handler* func) noexcept;
  int at_quick_exit(c-atexit-handler* func) noexcept;
  int at_quick_exit(atexit-handler* func) noexcept;
  [[noreturn]] void exit(int status);
  [[noreturn]] void _Exit(int status) noexcept;
  [[noreturn]] void quick_exit(int status) noexcept;

  char* getenv(const char* name);
  int system(const char* string);

  // ??, C library memory allocation
  void* aligned_alloc(size_t alignment, size_t size);
  void* calloc(size_t nmemb, size_t size);
  void free(void* ptr);
  void* malloc(size_t size);
  void* realloc(void* ptr, size_t size);

  double atof(const char* nptr);
  int atoi(const char* nptr);
  long int atol(const char* nptr);
```

```
  long long int atoll(const char* nptr);
  double strtod(const char* nptr, char** endptr);
  float strtof(const char* nptr, char** endptr);
  long double strtold(const char* nptr, char** endptr);
  long int strtol(const char* nptr, char** endptr, int base);
  long long int strtoll(const char* nptr, char** endptr, int base);
  unsigned long int strtoul(const char* nptr, char** endptr, int base);
  unsigned long long int strtoull(const char* nptr, char** endptr, int base);

  // ??, multibyte / wide string and character conversion functions
  int mblen(const char* s, size_t n);
  int mbtowc(wchar_t* pwc, const char* s, size_t n);
  int wctomb(char* s, wchar_t wchar);
  size_t mbstowcs(wchar_t* pwcs, const char* s, size_t n);
  size_t wcstombs(char* s, const wchar_t* pwcs, size_t n);

  // ??, C standard library algorithms
  void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
                c-compare-pred* compar);
  void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
                compare-pred* compar);
  void qsort(void* base, size_t nmemb, size_t size, c-compare-pred* compar);
  void qsort(void* base, size_t nmemb, size_t size, compare-pred* compar);

  // ??, low-quality random number generation
  int rand();
  void srand(unsigned int seed);

  // ??, absolute values
  int abs(int j);
  long int abs(long int j);
  long long int abs(long long int j);
  float abs(float j);
  double abs(double j);
  long double abs(long double j);

  long int labs(long int j);
  long long int llabs(long long int j);

  div_t div(int numer, int denom);
  ldiv_t div(long int numer, long int denom);             // see ??
  lldiv_t div(long long int numer, long long int denom);  // see ??
  ldiv_t ldiv(long int numer, long int denom);
  lldiv_t lldiv(long long int numer, long long int denom);
}
```

1 The contents and meaning of the header `<cstdlib>` are the same as the C standard library header `<stdlib.h>`, except that it does not declare the type `wchar_t`, and except as noted in 16.2.3, 16.2.4, 16.5, **??**, **??**, **??**, **??**, and **??**. [*Note*: Several functions have additional overloads in this document, but they have the same behavior as in the C standard library (**??**). — *end note*]

See also: ISO C 7.22

16.2.3 Null pointers [support.types.nullptr]

1 The type `nullptr_t` is a synonym for the type of a `nullptr` expression, and it has the characteristics described in **??** and **??**. [*Note*: Although `nullptr`'s address cannot be taken, the address of another `nullptr_t` object that is an lvalue can be taken. — *end note*]

2 The macro `NULL` is an implementation-defined null pointer constant.[187]

See also: ISO C 7.19

187) Possible definitions include `0` and `0L`, but not `(void*)0`.

16.2.4 Sizes, alignments, and offsets [support.types.layout]

1 The macro `offsetof(`*`type, member-designator`*`)` has the same semantics as the corresponding macro in the C standard library header `<stddef.h>`, but accepts a restricted set of *`type`* arguments in this document. Use of the `offsetof` macro with a *`type`* other than a standard-layout class (??) is conditionally-supported.[188] The expression `offsetof(`*`type, member-designator`*`)` is never type-dependent (??) and it is value-dependent (??) if and only if *`type`* is dependent. The result of applying the `offsetof` macro to a static data member or a function member is undefined. No operation invoked by the `offsetof` macro shall throw an exception and `noexcept(offsetof(`*`type, member-designator`*`))` shall be `true`.

2 The type `ptrdiff_t` is an implementation-defined signed integer type that can hold the difference of two subscripts in an array object, as described in ??.

3 The type `size_t` is an implementation-defined unsigned integer type that is large enough to contain the size in bytes of any object (??).

4 [*Note*: It is recommended that implementations choose types for `ptrdiff_t` and `size_t` whose integer conversion ranks (??) are no greater than that of `signed long int` unless a larger size is necessary to contain all the possible values. — *end note*]

5 The type `max_align_t` is a trivial standard-layout type whose alignment requirement is at least as great as that of every scalar type, and whose alignment requirement is supported in every context (??).

See also: ISO C 7.19

16.2.5 byte type operations [support.types.byteops]

```
template<class IntType>
  constexpr byte& operator<<=(byte& b, IntType shift) noexcept;
```

1 *Constraints:* `is_integral_v<IntType>` is `true`.

2 *Effects:* Equivalent to: `return b = b << shift;`

3 ~~*Remarks:* This function shall not participate in overload resolution unless `is_integral_v<IntType>` is `true`.~~

```
template<class IntType>
  constexpr byte operator<<(byte b, IntType shift) noexcept;
```

4 *Constraints:* `is_integral_v<IntType>` is `true`.

5 *Effects:* Equivalent to:

```
return static_cast<byte>(static_cast<unsigned char>(
                           static_cast<unsigned int>(b) << shift));
```

6 ~~*Remarks:* This function shall not participate in overload resolution unless `is_integral_v<IntType>` is `true`.~~

```
template<class IntType>
  constexpr byte& operator>>=(byte& b, IntType shift) noexcept;
```

7 *Constraints:* `is_integral_v<IntType>` is `true`.

8 *Effects:* Equivalent to: `return b >> shift;`

9 ~~*Remarks:* This function shall not participate in overload resolution unless `is_integral_v<IntType>` is `true`.~~

```
template<class IntType>
  constexpr byte operator>>(byte b, IntType shift) noexcept;
```

10 *Constraints:* `is_integral_v<IntType>` is `true`.

11 *Effects:* Equivalent to:

```
return static_cast<byte>(static_cast<unsigned char>(
                           static_cast<unsigned int>(b) >> shift));
```

12 ~~*Remarks:* This function shall not participate in overload resolution unless `is_integral_v<IntType>` is `true`.~~

188) Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.

```
constexpr byte& operator|=(byte& l, byte r) noexcept;
```

13 *Effects:* Equivalent to: `return l = l | r;`

```
constexpr byte operator|(byte l, byte r) noexcept;
```

14 *Effects:* Equivalent to:

```
return static_cast<byte>(static_cast<unsigned char>(static_cast<unsigned int>(l) |
                                                    static_cast<unsigned int>(r)));
```

```
constexpr byte& operator&=(byte& l, byte r) noexcept;
```

15 *Effects:* Equivalent to: `return l = l & r;`

```
constexpr byte operator&(byte l, byte r) noexcept;
```

16 *Effects:* Equivalent to:

```
return static_cast<byte>(static_cast<unsigned char>(static_cast<unsigned int>(l) &
                                                    static_cast<unsigned int>(r)));
```

```
constexpr byte& operator^=(byte& l, byte r) noexcept;
```

17 *Effects:* Equivalent to: `return l = l ^ r;`

```
constexpr byte operator^(byte l, byte r) noexcept;
```

18 *Effects:* Equivalent to:

```
return static_cast<byte>(static_cast<unsigned char>(static_cast<unsigned int>(l) ^
                                                    static_cast<unsigned int>(r)));
```

```
constexpr byte operator~(byte b) noexcept;
```

19 *Effects:* Equivalent to:

```
return static_cast<byte>(static_cast<unsigned char>(
                            ~static_cast<unsigned int>(b)));
```

```
template<class IntType>
  constexpr IntType to_integer(byte b) noexcept;
```

20 *Constraints:* `is_integral_v<IntType>` is `true`.

21 *Effects:* Equivalent to: `return static_cast<IntType>(b);`

22 ~~*Remarks:* This function shall not participate in overload resolution unless `is_integral_v<IntType>` is `true`.~~

16.3 Implementation properties [support.limits]

16.3.1 General [support.limits.general]

1 The headers `<limits>` (16.3.2), `<climits>` (16.3.5), and `<cfloat>` (16.3.6) supply characteristics of implementation-dependent arithmetic types (**??**).

2 The header `<version>` supplies implementation-dependent information about the C++ standard library (e.g., version number and release date).

3 The macros in Table 35 are defined after inclusion of the header `<version>` or one of the corresponding headers specified in the table. [*Note*: Future versions of this International Standard might replace the values of these macros with greater values. — *end note*]

Table 35 — Standard library feature-test macros

Macro name	Value	Header(s)
`__cpp_lib_addressof_constexpr`	201603L	`<memory>`
`__cpp_lib_allocator_traits_is_always_equal`	201411L	`<memory>` `<scoped_allocator>` `<string>` `<deque>` `<forward_list>` `<list>` `<vector>` `<map>` `<set>` `<unordered_map>` `<unordered_set>`

Table 35 — Standard library feature-test macros (continued)

Macro name	Value	Header(s)
__cpp_lib_any	201606L	<any>
__cpp_lib_apply	201603L	<tuple>
__cpp_lib_array_constexpr	201603L	<iterator> <array>
__cpp_lib_as_const	201510L	<utility>
__cpp_lib_atomic_is_always_lock_free	201603L	<atomic>
__cpp_lib_atomic_ref	201806L	<atomic>
__cpp_lib_bit_cast	201806L	<bit>
__cpp_lib_bind_front	201811L	<functional>
__cpp_lib_bool_constant	201505L	<type_traits>
__cpp_lib_boyer_moore_searcher	201603L	<functional>
__cpp_lib_byte	201603L	<cstddef>
__cpp_lib_char8_t	201811L	<atomic> <filesystem> <istream> <limits> <locale> <ostream> <string> <string_view>
__cpp_lib_chrono	201611L	<chrono>
__cpp_lib_chrono_udls	201304L	<chrono>
__cpp_lib_clamp	201603L	<algorithm>
__cpp_lib_complex_udls	201309L	<complex>
__cpp_lib_concepts	201806L	<concepts>
__cpp_lib_constexpr_misc	201811L	<array> <functional> <iterator> <string_view> <tuple> <utility>
__cpp_lib_constexpr_swap_algorithms	201806L	<algorithm>
__cpp_lib_destroying_delete	201806L	<new>
__cpp_lib_enable_shared_from_this	201603L	<memory>
__cpp_lib_erase_if	201811L	<string> <deque> <forward_list> <list> <vector> <map> <set> <unordered_map> <unordered_set>
__cpp_lib_exchange_function	201304L	<utility>
__cpp_lib_execution	201603L	<execution>
__cpp_lib_filesystem	201703L	<filesystem>
__cpp_lib_gcd_lcm	201606L	<numeric>
__cpp_lib_generic_associative_lookup	201304L	<map> <set>
__cpp_lib_generic_unordered_lookup	201811L	<unordered_map> <unordered_set>
__cpp_lib_hardware_interference_size	201703L	<new>
__cpp_lib_has_unique_object_representations	201606L	<type_traits>
__cpp_lib_hypot	201603L	<cmath>
__cpp_lib_incomplete_container_elements	201505L	<forward_list> <list> <vector>
__cpp_lib_integer_sequence	201304L	<utility>
__cpp_lib_integral_constant_callable	201304L	<type_traits>
__cpp_lib_invoke	201411L	<functional>
__cpp_lib_is_aggregate	201703L	<type_traits>
__cpp_lib_is_constant_evaluated	201811L	<type_traits>
__cpp_lib_is_final	201402L	<type_traits>
__cpp_lib_is_invocable	201703L	<type_traits>
__cpp_lib_is_null_pointer	201309L	<type_traits>
__cpp_lib_is_swappable	201603L	<type_traits>
__cpp_lib_launder	201606L	<new>
__cpp_lib_list_remove_return_type	201806L	<forward_list> <list>

Table 35 — Standard library feature-test macros (continued)

Macro name	Value	Header(s)
__cpp_lib_logical_traits	201510L	<type_traits>
__cpp_lib_make_from_tuple	201606L	<tuple>
__cpp_lib_make_reverse_iterator	201402L	<iterator>
__cpp_lib_make_unique	201304L	<memory>
__cpp_lib_map_try_emplace	201411L	<map>
__cpp_lib_math_special_functions	201603L	<cmath>
__cpp_lib_memory_resource	201603L	<memory_resource>
__cpp_lib_node_extract	201606L	<map> <set> <unordered_map> <unordered_set>
__cpp_lib_nonmember_container_access	201411L	<iterator> <array> <deque> <forward_list> <list> <map> <regex> <set> <string> <unordered_map> <unordered_set> <vector>
__cpp_lib_not_fn	201603L	<functional>
__cpp_lib_null_iterators	201304L	<iterator>
__cpp_lib_optional	201606L	<optional>
__cpp_lib_parallel_algorithm	201603L	<algorithm> <numeric>
__cpp_lib_quoted_string_io	201304L	<iomanip>
__cpp_lib_ranges	201811L	<algorithm> <functional> <iterator> <memory> <ranges>
__cpp_lib_raw_memory_algorithms	201606L	<memory>
__cpp_lib_result_of_sfinae	201210L	<functional> <type_traits>
__cpp_lib_robust_nonmodifying_seq_ops	201304L	<algorithm>
__cpp_lib_sample	201603L	<algorithm>
__cpp_lib_scoped_lock	201703L	<mutex>
__cpp_lib_shared_mutex	201505L	<shared_mutex>
__cpp_lib_shared_ptr_arrays	201611L	<memory>
__cpp_lib_shared_ptr_weak_type	201606L	<memory>
__cpp_lib_shared_timed_mutex	201402L	<shared_mutex>
__cpp_lib_string_udls	201304L	<string>
__cpp_lib_string_view	201606L	<string> <string_view>
__cpp_lib_three_way_comparison	201711L	<compare>
__cpp_lib_to_chars	201611L	<charconv>
__cpp_lib_transformation_trait_aliases	201304L	<type_traits>
__cpp_lib_transparent_operators	201510L	<memory> <functional>
__cpp_lib_tuple_element_t	201402L	<tuple>
__cpp_lib_tuples_by_type	201304L	<utility> <tuple>
__cpp_lib_type_trait_variable_templates	201510L	<type_traits>
__cpp_lib_uncaught_exceptions	201411L	<exception>
__cpp_lib_unordered_map_try_emplace	201411L	<unordered_map>
__cpp_lib_variant	201606L	<variant>
__cpp_lib_void_t	201411L	<type_traits>

16.3.2 Header <limits> synopsis [limits.syn]

```
namespace std {
  // 16.3.3, floating-point type properties
  enum float_round_style;
  enum float_denorm_style;

  // 16.3.4, class template numeric_limits
  template<class T> class numeric_limits;

  template<> class numeric_limits<bool>;
```

```
  template<> class numeric_limits<char>;
  template<> class numeric_limits<signed char>;
  template<> class numeric_limits<unsigned char>;
  template<> class numeric_limits<char8_t>;
  template<> class numeric_limits<char16_t>;
  template<> class numeric_limits<char32_t>;
  template<> class numeric_limits<wchar_t>;

  template<> class numeric_limits<short>;
  template<> class numeric_limits<int>;
  template<> class numeric_limits<long>;
  template<> class numeric_limits<long long>;
  template<> class numeric_limits<unsigned short>;
  template<> class numeric_limits<unsigned int>;
  template<> class numeric_limits<unsigned long>;
  template<> class numeric_limits<unsigned long long>;

  template<> class numeric_limits<float>;
  template<> class numeric_limits<double>;
  template<> class numeric_limits<long double>;
}
```

16.3.3 Floating-point type properties [fp.style]

16.3.3.1 Type `float_round_style` [round.style]

```
namespace std {
  enum float_round_style {
    round_indeterminate       = -1,
    round_toward_zero         =  0,
    round_to_nearest          =  1,
    round_toward_infinity     =  2,
    round_toward_neg_infinity =  3
  };
}
```

1 The rounding mode for floating-point arithmetic is characterized by the values:

(1.1) — `round_indeterminate` if the rounding style is indeterminable

(1.2) — `round_toward_zero` if the rounding style is toward zero

(1.3) — `round_to_nearest` if the rounding style is to the nearest representable value

(1.4) — `round_toward_infinity` if the rounding style is toward infinity

(1.5) — `round_toward_neg_infinity` if the rounding style is toward negative infinity

16.3.3.2 Type `float_denorm_style` [denorm.style]

```
namespace std {
  enum float_denorm_style {
    denorm_indeterminate = -1,
    denorm_absent = 0,
    denorm_present = 1
  };
}
```

1 The presence or absence of subnormal numbers (variable number of exponent bits) is characterized by the values:

(1.1) — `denorm_indeterminate` if it cannot be determined whether or not the type allows subnormal values

(1.2) — `denorm_absent` if the type does not allow subnormal values

(1.3) — `denorm_present` if the type does allow subnormal values

16.3.4 Class template `numeric_limits` [numeric.limits]

1 The `numeric_limits` class template provides a C++ program with information about various properties of the implementation's representation of the arithmetic types.

```
namespace std {
  template<class T> class numeric_limits {
  public:
    static constexpr bool is_specialized = false;
    static constexpr T min() noexcept { return T(); }
    static constexpr T max() noexcept { return T(); }
    static constexpr T lowest() noexcept { return T(); }

    static constexpr int  digits = 0;
    static constexpr int  digits10 = 0;
    static constexpr int  max_digits10 = 0;
    static constexpr bool is_signed = false;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact = false;
    static constexpr int  radix = 0;
    static constexpr T epsilon() noexcept { return T(); }
    static constexpr T round_error() noexcept { return T(); }

    static constexpr int  min_exponent = 0;
    static constexpr int  min_exponent10 = 0;
    static constexpr int  max_exponent = 0;
    static constexpr int  max_exponent10 = 0;

    static constexpr bool has_infinity = false;
    static constexpr bool has_quiet_NaN = false;
    static constexpr bool has_signaling_NaN = false;
    static constexpr float_denorm_style has_denorm = denorm_absent;
    static constexpr bool has_denorm_loss = false;
    static constexpr T infinity() noexcept { return T(); }
    static constexpr T quiet_NaN() noexcept { return T(); }
    static constexpr T signaling_NaN() noexcept { return T(); }
    static constexpr T denorm_min() noexcept { return T(); }

    static constexpr bool is_iec559 = false;
    static constexpr bool is_bounded = false;
    static constexpr bool is_modulo = false;

    static constexpr bool traps = false;
    static constexpr bool tinyness_before = false;
    static constexpr float_round_style round_style = round_toward_zero;
  };

  template<class T> class numeric_limits<const T>;
  template<class T> class numeric_limits<volatile T>;
  template<class T> class numeric_limits<const volatile T>;
}
```

2 For all members declared `static constexpr` in the `numeric_limits` template, specializations shall define these values in such a way that they are usable as constant expressions.

3 The default `numeric_limits<T>` template shall have all members, but with 0 or `false` values.

4 Specializations shall be provided for each arithmetic type, both floating-point and integer, including `bool`. The member `is_specialized` shall be `true` for all such specializations of `numeric_limits`.

5 The value of each member of a specialization of `numeric_limits` on a cv-qualified type `cv T` shall be equal to the value of the corresponding member of the specialization on the unqualified type `T`.

6 Non-arithmetic standard types, such as `complex<T>` (**??**), shall not have specializations.

16.3.4.1 `numeric_limits` members [numeric.limits.members]

1 Each member function defined in this subclause is signal-safe (16.12.4).

`static constexpr T min() noexcept;`

Minimum finite value.[189]

For floating-point types with subnormal numbers, returns the minimum positive normalized value.

Meaningful for all specializations in which `is_bounded != false`, or `is_bounded == false && is_signed == false`.

`static constexpr T max() noexcept;`

Maximum finite value.[190]

Meaningful for all specializations in which `is_bounded != false`.

`static constexpr T lowest() noexcept;`

A finite value `x` such that there is no other finite value `y` where `y < x`.[191]

Meaningful for all specializations in which `is_bounded != false`.

`static constexpr int digits;`

Number of `radix` digits that can be represented without change.

For integer types, the number of non-sign bits in the representation.

For floating-point types, the number of `radix` digits in the mantissa.[192]

`static constexpr int digits10;`

Number of base 10 digits that can be represented without change.[193]

Meaningful for all specializations in which `is_bounded != false`.

`static constexpr int max_digits10;`

Number of base 10 digits required to ensure that values which differ are always differentiated.

Meaningful for all floating-point types.

`static constexpr bool is_signed;`

`true` if the type is signed.

Meaningful for all specializations.

`static constexpr bool is_integer;`

`true` if the type is integer.

Meaningful for all specializations.

`static constexpr bool is_exact;`

`true` if the type uses an exact representation. All integer types are exact, but not all exact types are integer. For example, rational and fixed-exponent representations are exact but not integer.

Meaningful for all specializations.

`static constexpr int radix;`

For floating-point types, specifies the base or radix of the exponent representation (often 2).[194]

For integer types, specifies the base of the representation.[195]

Meaningful for all specializations.

189) Equivalent to `CHAR_MIN`, `SHRT_MIN`, `FLT_MIN`, `DBL_MIN`, etc.
190) Equivalent to `CHAR_MAX`, `SHRT_MAX`, `FLT_MAX`, `DBL_MAX`, etc.
191) `lowest()` is necessary because not all floating-point representations have a smallest (most negative) value that is the negative of the largest (most positive) finite value.
192) Equivalent to `FLT_MANT_DIG`, `DBL_MANT_DIG`, `LDBL_MANT_DIG`.
193) Equivalent to `FLT_DIG`, `DBL_DIG`, `LDBL_DIG`.
194) Equivalent to `FLT_RADIX`.
195) Distinguishes types with bases other than 2 (e.g. BCD).

`static constexpr T epsilon() noexcept;`

25 Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.[196]

26 Meaningful for all floating-point types.

`static constexpr T round_error() noexcept;`

27 Measure of the maximum rounding error.[197]

`static constexpr int  min_exponent;`

28 Minimum negative integer such that `radix` raised to the power of one less than that integer is a normalized floating-point number.[198]

29 Meaningful for all floating-point types.

`static constexpr int  min_exponent10;`

30 Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.[199]

31 Meaningful for all floating-point types.

`static constexpr int  max_exponent;`

32 Maximum positive integer such that `radix` raised to the power one less than that integer is a representable finite floating-point number.[200]

33 Meaningful for all floating-point types.

`static constexpr int  max_exponent10;`

34 Maximum positive integer such that 10 raised to that power is in the range of representable finite floating-point numbers.[201]

35 Meaningful for all floating-point types.

`static constexpr bool has_infinity;`

36 `true` if the type has a representation for positive infinity.

37 Meaningful for all floating-point types.

38 Shall be `true` for all specializations in which `is_iec559 != false`.

`static constexpr bool has_quiet_NaN;`

39 `true` if the type has a representation for a quiet (non-signaling) "Not a Number".[202]

40 Meaningful for all floating-point types.

41 Shall be `true` for all specializations in which `is_iec559 != false`.

`static constexpr bool has_signaling_NaN;`

42 `true` if the type has a representation for a signaling "Not a Number".[203]

43 Meaningful for all floating-point types.

44 Shall be `true` for all specializations in which `is_iec559 != false`.

`static constexpr float_denorm_style has_denorm;`

45 `denorm_present` if the type allows subnormal values (variable number of exponent bits)[204], `denorm_absent` if the type does not allow subnormal values, and `denorm_indeterminate` if it is indeterminate at compile time whether the type allows subnormal values.

196) Equivalent to `FLT_EPSILON`, `DBL_EPSILON`, `LDBL_EPSILON`.
197) Rounding error is described in LIA-1 Section 5.2.4 and Annex C Rationale Section C.5.2.4 — Rounding and rounding constants.
198) Equivalent to `FLT_MIN_EXP`, `DBL_MIN_EXP`, `LDBL_MIN_EXP`.
199) Equivalent to `FLT_MIN_10_EXP`, `DBL_MIN_10_EXP`, `LDBL_MIN_10_EXP`.
200) Equivalent to `FLT_MAX_EXP`, `DBL_MAX_EXP`, `LDBL_MAX_EXP`.
201) Equivalent to `FLT_MAX_10_EXP`, `DBL_MAX_10_EXP`, `LDBL_MAX_10_EXP`.
202) Required by LIA-1.
203) Required by LIA-1.
204) Required by LIA-1.

46 Meaningful for all floating-point types.

`static constexpr bool has_denorm_loss;`

47 `true` if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.[205]

`static constexpr T infinity() noexcept;`

48 Representation of positive infinity, if available.[206]

49 Meaningful for all specializations for which `has_infinity != false`. Required in specializations for which `is_iec559 != false`.

`static constexpr T quiet_NaN() noexcept;`

50 Representation of a quiet "Not a Number", if available.[207]

51 Meaningful for all specializations for which `has_quiet_NaN != false`. Required in specializations for which `is_iec559 != false`.

`static constexpr T signaling_NaN() noexcept;`

52 Representation of a signaling "Not a Number", if available.[208]

53 Meaningful for all specializations for which `has_signaling_NaN != false`. Required in specializations for which `is_iec559 != false`.

`static constexpr T denorm_min() noexcept;`

54 Minimum positive subnormal value.[209]

55 Meaningful for all floating-point types.

56 In specializations for which `has_denorm == false`, returns the minimum positive normalized value.

`static constexpr bool is_iec559;`

57 `true` if and only if the type adheres to ISO/IEC/IEEE 60559.[210]

58 Meaningful for all floating-point types.

`static constexpr bool is_bounded;`

59 `true` if the set of values representable by the type is finite.[211] [*Note*: All fundamental types (**??**) are bounded. This member would be `false` for arbitrary precision types. — *end note*]

60 Meaningful for all specializations.

`static constexpr bool is_modulo;`

61 `true` if the type is modulo.[212] A type is modulo if, for any operation involving `+`, `-`, or `*` on values of that type whose result would fall outside the range `[min(), max()]`, the value returned differs from the true value by an integer multiple of `max() - min() + 1`.

62 [*Example*: `is_modulo` is `false` for signed integer types (**??**) unless an implementation, as an extension to this document, defines signed integer overflow to wrap. — *end example*]

63 Meaningful for all specializations.

`static constexpr bool traps;`

64 `true` if, at program startup, there exists a value of the type that would cause an arithmetic operation using that value to trap.[213]

65 Meaningful for all specializations.

205) See ISO/IEC/IEEE 60559.
206) Required by LIA-1.
207) Required by LIA-1.
208) Required by LIA-1.
209) Required by LIA-1.
210) ISO/IEC/IEEE 60559:2011 is the same as IEEE 754-2008.
211) Required by LIA-1.
212) Required by LIA-1.
213) Required by LIA-1.

`static constexpr bool tinyness_before;`

66 `true` if tinyness is detected before rounding.[214]

67 Meaningful for all floating-point types.

`static constexpr float_round_style round_style;`

68 The rounding style for the type.[215]

69 Meaningful for all floating-point types. Specializations for integer types shall return `round_toward_zero`.

16.3.4.2 `numeric_limits` specializations [numeric.special]

1 All members shall be provided for all specializations. However, many values are only required to be meaningful under certain conditions (for example, `epsilon()` is only meaningful if `is_integer` is `false`). Any value that is not "meaningful" shall be set to 0 or `false`.

2 [*Example*:

```
namespace std {
  template<> class numeric_limits<float> {
  public:
    static constexpr bool is_specialized = true;

    static constexpr float min() noexcept { return 1.17549435E-38F; }
    static constexpr float max() noexcept { return 3.40282347E+38F; }
    static constexpr float lowest() noexcept { return -3.40282347E+38F; }

    static constexpr int digits   = 24;
    static constexpr int digits10 =  6;
    static constexpr int max_digits10 =  9;

    static constexpr bool is_signed  = true;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact   = false;

    static constexpr int radix = 2;
    static constexpr float epsilon() noexcept     { return 1.19209290E-07F; }
    static constexpr float round_error() noexcept { return 0.5F; }

    static constexpr int min_exponent   = -125;
    static constexpr int min_exponent10 = - 37;
    static constexpr int max_exponent   = +128;
    static constexpr int max_exponent10 = + 38;

    static constexpr bool has_infinity             = true;
    static constexpr bool has_quiet_NaN            = true;
    static constexpr bool has_signaling_NaN        = true;
    static constexpr float_denorm_style has_denorm = denorm_absent;
    static constexpr bool has_denorm_loss          = false;

    static constexpr float infinity()      noexcept { return value; }
    static constexpr float quiet_NaN()     noexcept { return value; }
    static constexpr float signaling_NaN() noexcept { return value; }
    static constexpr float denorm_min()    noexcept { return min(); }

    static constexpr bool is_iec559  = true;
    static constexpr bool is_bounded = true;
    static constexpr bool is_modulo  = false;
    static constexpr bool traps      = true;
    static constexpr bool tinyness_before = true;
```

214) Refer to ISO/IEC/IEEE 60559. Required by LIA-1.
215) Equivalent to `FLT_ROUNDS`. Required by LIA-1.

```
    static constexpr float_round_style round_style = round_to_nearest;
  };
}
```

— *end example*]

3 The specialization for `bool` shall be provided as follows:

```
namespace std {
   template<> class numeric_limits<bool> {
   public:
     static constexpr bool is_specialized = true;
     static constexpr bool min() noexcept { return false; }
     static constexpr bool max() noexcept { return true; }
     static constexpr bool lowest() noexcept { return false; }

     static constexpr int  digits = 1;
     static constexpr int  digits10 = 0;
     static constexpr int  max_digits10 = 0;

     static constexpr bool is_signed = false;
     static constexpr bool is_integer = true;
     static constexpr bool is_exact = true;
     static constexpr int  radix = 2;
     static constexpr bool epsilon() noexcept { return 0; }
     static constexpr bool round_error() noexcept { return 0; }

     static constexpr int  min_exponent = 0;
     static constexpr int  min_exponent10 = 0;
     static constexpr int  max_exponent = 0;
     static constexpr int  max_exponent10 = 0;

     static constexpr bool has_infinity = false;
     static constexpr bool has_quiet_NaN = false;
     static constexpr bool has_signaling_NaN = false;
     static constexpr float_denorm_style has_denorm = denorm_absent;
     static constexpr bool has_denorm_loss = false;
     static constexpr bool infinity() noexcept { return 0; }
     static constexpr bool quiet_NaN() noexcept { return 0; }
     static constexpr bool signaling_NaN() noexcept { return 0; }
     static constexpr bool denorm_min() noexcept { return 0; }

     static constexpr bool is_iec559 = false;
     static constexpr bool is_bounded = true;
     static constexpr bool is_modulo = false;

     static constexpr bool traps = false;
     static constexpr bool tinyness_before = false;
     static constexpr float_round_style round_style = round_toward_zero;
   };
}
```

16.3.5 Header <climits> synopsis [climits.syn]

```
#define CHAR_BIT see below
#define SCHAR_MIN see below
#define SCHAR_MAX see below
#define UCHAR_MAX see below
#define CHAR_MIN see below
#define CHAR_MAX see below
#define MB_LEN_MAX see below
#define SHRT_MIN see below
#define SHRT_MAX see below
#define USHRT_MAX see below
#define INT_MIN see below
#define INT_MAX see below
```

```
#define UINT_MAX see below
#define LONG_MIN see below
#define LONG_MAX see below
#define ULONG_MAX see below
#define LLONG_MIN see below
#define LLONG_MAX see below
#define ULLONG_MAX see below
```

1 The header <climits> defines all macros the same as the C standard library header <limits.h>. [*Note*: The types of the constants defined by macros in <climits> are not required to match the types to which the macros refer. — *end note*]

See also: ISO C 5.2.4.2.1

16.3.6 Header <cfloat> synopsis [cfloat.syn]

```
#define FLT_ROUNDS see below
#define FLT_EVAL_METHOD see below
#define FLT_HAS_SUBNORM see below
#define DBL_HAS_SUBNORM see below
#define LDBL_HAS_SUBNORM see below
#define FLT_RADIX see below
#define FLT_MANT_DIG see below
#define DBL_MANT_DIG see below
#define LDBL_MANT_DIG see below
#define FLT_DECIMAL_DIG see below
#define DBL_DECIMAL_DIG see below
#define LDBL_DECIMAL_DIG see below
#define DECIMAL_DIG see below
#define FLT_DIG see below
#define DBL_DIG see below
#define LDBL_DIG see below
#define FLT_MIN_EXP see below
#define DBL_MIN_EXP see below
#define LDBL_MIN_EXP see below
#define FLT_MIN_10_EXP see below
#define DBL_MIN_10_EXP see below
#define LDBL_MIN_10_EXP see below
#define FLT_MAX_EXP see below
#define DBL_MAX_EXP see below
#define LDBL_MAX_EXP see below
#define FLT_MAX_10_EXP see below
#define DBL_MAX_10_EXP see below
#define LDBL_MAX_10_EXP see below
#define FLT_MAX see below
#define DBL_MAX see below
#define LDBL_MAX see below
#define FLT_EPSILON see below
#define DBL_EPSILON see below
#define LDBL_EPSILON see below
#define FLT_MIN see below
#define DBL_MIN see below
#define LDBL_MIN see below
#define FLT_TRUE_MIN see below
#define DBL_TRUE_MIN see below
#define LDBL_TRUE_MIN see below
```

1 The header <cfloat> defines all macros the same as the C standard library header <float.h>.

See also: ISO C 5.2.4.2.2

16.4 Integer types [cstdint]

16.4.1 Header <cstdint> synopsis [cstdint.syn]

```
namespace std {
  using int8_t         = signed integer type;  // optional
  using int16_t        = signed integer type;  // optional
```

```
  using int32_t        = signed integer type;  // optional
  using int64_t        = signed integer type;  // optional

  using int_fast8_t    = signed integer type;
  using int_fast16_t   = signed integer type;
  using int_fast32_t   = signed integer type;
  using int_fast64_t   = signed integer type;

  using int_least8_t   = signed integer type;
  using int_least16_t  = signed integer type;
  using int_least32_t  = signed integer type;
  using int_least64_t  = signed integer type;

  using intmax_t       = signed integer type;
  using intptr_t       = signed integer type;   // optional

  using uint8_t        = unsigned integer type; // optional
  using uint16_t       = unsigned integer type; // optional
  using uint32_t       = unsigned integer type; // optional
  using uint64_t       = unsigned integer type; // optional

  using uint_fast8_t   = unsigned integer type;
  using uint_fast16_t  = unsigned integer type;
  using uint_fast32_t  = unsigned integer type;
  using uint_fast64_t  = unsigned integer type;

  using uint_least8_t  = unsigned integer type;
  using uint_least16_t = unsigned integer type;
  using uint_least32_t = unsigned integer type;
  using uint_least64_t = unsigned integer type;

  using uintmax_t      = unsigned integer type;
  using uintptr_t      = unsigned integer type; // optional
}
```

1 The header also defines numerous macros of the form:

```
INT_[FAST LEAST]{8 16 32 64}_MIN
[U]INT_[FAST LEAST]{8 16 32 64}_MAX
INT{MAX PTR}_MIN
[U]INT{MAX PTR}_MAX
{PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
SIZE_MAX
```

plus function macros of the form:

```
[U]INT{8 16 32 64 MAX}_C
```

2 The header defines all types and macros the same as the C standard library header `<stdint.h>`.

See also: ISO C 7.20

16.5 Start and termination [support.start.term]

1 [*Note*: The header `<cstdlib>` (16.2.2) declares the functions described in this subclause. — *end note*]

```
[[noreturn]] void _Exit(int status) noexcept;
```

2 *Effects:* This function has the semantics specified in the C standard library.

3 *Remarks:* The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()` (??). The function `_Exit` is signal-safe (16.12.4).

```
[[noreturn]] void abort() noexcept;
```

4 *Effects:* This function has the semantics specified in the C standard library.

5 *Remarks:* The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()` (**??**). The function `abort` is signal-safe (16.12.4).

```
int atexit(c-atexit-handler* f) noexcept;
int atexit(atexit-handler* f) noexcept;
```

6 *Effects:* The `atexit()` functions register the function pointed to by `f` to be called without arguments at normal program termination. It is unspecified whether a call to `atexit()` that does not happen before (**??**) a call to `exit()` will succeed. [*Note*: The `atexit()` functions do not introduce a data race (**??**). — *end note*]

7 *Implementation limits:* The implementation shall support the registration of at least 32 functions.

8 *Returns:* The `atexit()` function returns zero if the registration succeeds, nonzero if it fails.

```
[[noreturn]] void exit(int status);
```

9 *Effects:*

(9.1) — First, objects with thread storage duration and associated with the current thread are destroyed. Next, objects with static storage duration are destroyed and functions registered by calling `atexit` are called.[216] See **??** for the order of destructions and calls. (Automatic objects are not destroyed as a result of calling `exit()`.)[217]

If control leaves a registered function called by `exit` because the function does not provide a handler for a thrown exception, the function `std::terminate` shall be called (**??**).

(9.2) — Next, all open C streams (as mediated by the function signatures declared in `<cstdio>`) with unwritten buffered data are flushed, all open C streams are closed, and all files created by calling `tmpfile()` are removed.

(9.3) — Finally, control is returned to the host environment. If `status` is zero or `EXIT_SUCCESS`, an implementation-defined form of the status *successful termination* is returned. If `status` is `EXIT_FAILURE`, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.[218]

```
int at_quick_exit(c-atexit-handler* f) noexcept;
int at_quick_exit(atexit-handler* f) noexcept;
```

10 *Effects:* The `at_quick_exit()` functions register the function pointed to by `f` to be called without arguments when `quick_exit` is called. It is unspecified whether a call to `at_quick_exit()` that does not happen before (**??**) all calls to `quick_exit` will succeed. [*Note*: The `at_quick_exit()` functions do not introduce a data race (**??**). — *end note*] [*Note*: The order of registration may be indeterminate if `at_quick_exit` was called from more than one thread. — *end note*] [*Note*: The `at_quick_exit` registrations are distinct from the `atexit` registrations, and applications may need to call both registration functions with the same argument. — *end note*]

11 *Implementation limits:* The implementation shall support the registration of at least 32 functions.

12 *Returns:* Zero if the registration succeeds, nonzero if it fails.

```
[[noreturn]] void quick_exit(int status) noexcept;
```

13 *Effects:* Functions registered by calls to `at_quick_exit` are called in the reverse order of their registration, except that a function shall be called after any previously registered functions that had already been called at the time it was registered. Objects shall not be destroyed as a result of calling `quick_exit`. If control leaves a registered function called by `quick_exit` because the function does not provide a handler for a thrown exception, the function `std::terminate` shall be called. [*Note*: A function registered via `at_quick_exit` is invoked by the thread that calls `quick_exit`, which can be a different thread than the one that registered it, so registered functions should not rely on the identity of objects with thread storage duration. — *end note*] After calling registered functions, `quick_exit` shall call `_Exit(status)`.

---

216) A function is called for every time it is registered.

217) Objects with automatic storage duration are all destroyed in a program whose `main` function (**??**) contains no automatic objects and executes the call to `exit()`. Control can be transferred directly to such a `main` function by throwing an exception that is caught in `main`.

218) The macros `EXIT_FAILURE` and `EXIT_SUCCESS` are defined in `<cstdlib>`.

14 *Remarks:* The function `quick_exit` is signal-safe (16.12.4) when the functions registered with `at_quick_exit` are.

See also: ISO C 7.22.4

16.6 Dynamic memory management [support.dynamic]

1 The header `<new>` defines several functions that manage the allocation of dynamic storage in a program. It also defines components for reporting storage management errors.

16.6.1 Header `<new>` synopsis [new.syn]

```
namespace std {
  class bad_alloc;
  class bad_array_new_length;

  struct destroying_delete_t {
    explicit destroying_delete_t() = default;
  };
  inline constexpr destroying_delete_t destroying_delete{};

  enum class align_val_t : size_t {};

  struct nothrow_t { explicit nothrow_t() = default; };
  extern const nothrow_t nothrow;

  using new_handler = void (*)();
  new_handler get_new_handler() noexcept;
  new_handler set_new_handler(new_handler new_p) noexcept;

  // 16.6.4, pointer optimization barrier
  template<class T> [[nodiscard]] constexpr T* launder(T* p) noexcept;

  // 16.6.5, hardware interference size
  inline constexpr size_t hardware_destructive_interference_size = implementation-defined;
  inline constexpr size_t hardware_constructive_interference_size = implementation-defined;
}

[[nodiscard]] void* operator new(std::size_t size);
[[nodiscard]] void* operator new(std::size_t size, std::align_val_t alignment);
[[nodiscard]] void* operator new(std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new(std::size_t size, std::align_val_t alignment,
                                 const std::nothrow_t&) noexcept;

void  operator delete(void* ptr) noexcept;
void  operator delete(void* ptr, std::size_t size) noexcept;
void  operator delete(void* ptr, std::align_val_t alignment) noexcept;
void  operator delete(void* ptr, std::size_t size, std::align_val_t alignment) noexcept;
void  operator delete(void* ptr, const std::nothrow_t&) noexcept;
void  operator delete(void* ptr, std::align_val_t alignment, const std::nothrow_t&) noexcept;

[[nodiscard]] void* operator new[](std::size_t size);
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment);
[[nodiscard]] void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment,
                                   const std::nothrow_t&) noexcept;

void  operator delete[](void* ptr) noexcept;
void  operator delete[](void* ptr, std::size_t size) noexcept;
void  operator delete[](void* ptr, std::align_val_t alignment) noexcept;
void  operator delete[](void* ptr, std::size_t size, std::align_val_t alignment) noexcept;
void  operator delete[](void* ptr, const std::nothrow_t&) noexcept;
void  operator delete[](void* ptr, std::align_val_t alignment, const std::nothrow_t&) noexcept;
```

```
[[nodiscard]] void* operator new  (std::size_t size, void* ptr) noexcept;
[[nodiscard]] void* operator new[](std::size_t size, void* ptr) noexcept;
void  operator delete  (void* ptr, void*) noexcept;
void  operator delete[](void* ptr, void*) noexcept;
```

16.6.2 Storage allocation and deallocation [new.delete]

1 Except where otherwise specified, the provisions of **??** apply to the library versions of `operator new` and `operator delete`. If the value of an alignment argument passed to any of these functions is not a valid alignment value, the behavior is undefined.

16.6.2.1 Single-object forms [new.delete.single]

```
[[nodiscard]] void* operator new(std::size_t size);
[[nodiscard]] void* operator new(std::size_t size, std::align_val_t alignment);
```

1 *Effects:* The allocation functions (**??**) called by a *new-expression* (**??**) to allocate `size` bytes of storage. The second form is called for a type with new-extended alignment, and allocates storage with the specified alignment. The first form is called otherwise, and allocates storage suitably aligned to represent any object of that size provided the object's type does not have new-extended alignment.

2 *Replaceable:* A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

3 *Required behavior:* Return a non-null pointer to suitably aligned storage (**??**), or else throw a `bad_alloc` exception. This requirement is binding on any replacement versions of these functions.

4 *Default behavior:*

(4.1) — Executes a loop: Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the C standard library functions `malloc` or `aligned_alloc` is unspecified.

(4.2) — Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the current `new_handler` (16.6.3.5) is a null pointer value, throws `bad_alloc`.

(4.3) — Otherwise, the function calls the current `new_handler` function (16.6.3.3). If the called function returns, the loop repeats.

(4.4) — The loop terminates when an attempt to allocate the requested storage is successful or when a called `new_handler` function does not return.

```
[[nodiscard]] void* operator new(std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new(std::size_t size, std::align_val_t alignment,
                                 const std::nothrow_t&) noexcept;
```

5 *Effects:* Same as above, except that these are called by a placement version of a *new-expression* when a C++ program prefers a null pointer result as an error indication, instead of a `bad_alloc` exception.

6 *Replaceable:* A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

7 *Required behavior:* Return a non-null pointer to suitably aligned storage (**??**), or else return a null pointer. Each of these nothrow versions of `operator new` returns a pointer obtained as if acquired from the (possibly replaced) corresponding non-placement function. This requirement is binding on any replacement versions of these functions.

8 *Default behavior:* Calls `operator new(size)`, or `operator new(size, alignment)`, respectively. If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

9 [*Example*:

```
T* p1 = new T;                  // throws bad_alloc if it fails
T* p2 = new(nothrow) T;         // returns nullptr if it fails
```

— *end example*]

```
void operator delete(void* ptr) noexcept;
void operator delete(void* ptr, std::size_t size) noexcept;
void operator delete(void* ptr, std::align_val_t alignment) noexcept;
```

```
void operator delete(void* ptr, std::size_t size, std::align_val_t alignment) noexcept;
```

10 *Effects:* The deallocation functions (**??**) called by a *delete-expression* (**??**) to render the value of `ptr` invalid.

11 *Replaceable:* A C++ program may define functions with any of these function signatures, and thereby displace the default versions defined by the C++ standard library. If a function without a `size` parameter is defined, the program should also define the corresponding function with a `size` parameter. If a function with a `size` parameter is defined, the program shall also define the corresponding version without the `size` parameter. [*Note*: The default behavior below may change in the future, which will require replacing both deallocation functions when replacing the allocation function. — *end note*]

12 ~~*Requires:*~~ *Expects:* `ptr` ~~shall be a null pointer or its value shall represent~~is a null pointer or its value represents the address of a block of memory allocated by an earlier call to a (possibly replaced) `operator new(std::size_t)` or `operator new(std::size_t, std::align_val_t)` which has not been invalidated by an intervening call to `operator delete`.

13 ~~*Requires:*~~ *Expects:* If an implementation has strict pointer safety (**??**) then `ptr` ~~shall be~~is a safely-derived pointer.

14 ~~*Requires:*~~ *Expects:* If the `alignment` parameter is not present, `ptr` ~~shall have been~~was returned by an allocation function without an `alignment` parameter. If present, the `alignment` argument ~~shall equal~~is equal to the `alignment` argument passed to the allocation function that returned `ptr`. If present, the `size` argument ~~shall equal~~is equal to the `size` argument passed to the allocation function that returned `ptr`.

15 *Required behavior:* A call to an `operator delete` with a `size` parameter may be changed to a call to the corresponding `operator delete` without a `size` parameter, without affecting memory allocation. [*Note*: A conforming implementation is for `operator delete(void* ptr, std::size_t size)` to simply call `operator delete(ptr)`. — *end note*]

16 *Default behavior:* The functions that have a `size` parameter forward their other parameters to the corresponding function without a `size` parameter. [*Note*: See the note in the above *Replaceable:* paragraph. — *end note*]

17 *Default behavior:* If `ptr` is null, does nothing. Otherwise, reclaims the storage allocated by the earlier call to `operator new`.

18 *Remarks:* It is unspecified under what conditions part or all of such reclaimed storage will be allocated by subsequent calls to `operator new` or any of `aligned_alloc`, `calloc`, `malloc`, or `realloc`, declared in `<cstdlib>`.

```
void operator delete(void* ptr, const std::nothrow_t&) noexcept;
void operator delete(void* ptr, std::align_val_t alignment, const std::nothrow_t&) noexcept;
```

19 *Effects:* The deallocation functions (**??**) called by the implementation to render the value of `ptr` invalid when the constructor invoked from a nothrow placement version of the *new-expression* throws an exception.

20 *Replaceable:* A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

21 ~~*Requires:*~~ *Expects:* `ptr` ~~shall be a null pointer or its value shall represent~~is a null pointer or its value represents the address of a block of memory allocated by an earlier call to a (possibly replaced) `operator new(std::size_t)` or `operator new(std::size_t, std::align_val_t)` which has not been invalidated by an intervening call to `operator delete`.

22 ~~*Requires:*~~ *Expects:* If an implementation has strict pointer safety (**??**) then `ptr` ~~shall be~~is a safely-derived pointer.

23 ~~*Requires:*~~ *Expects:* If the `alignment` parameter is not present, `ptr` ~~shall have been~~was returned by an allocation function without an `alignment` parameter. If present, the `alignment` argument ~~shall equal~~is equal to the `alignment` argument passed to the allocation function that returned `ptr`.

24 *Default behavior:* Calls `operator delete(ptr)`, or `operator delete(ptr, alignment)`, respectively.

16.6.2.2 Array forms [new.delete.array]

```
[[nodiscard]] void* operator new[](std::size_t size);
```

```
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment);
```

1 *Effects:* The allocation functions (**??**) called by the array form of a *new-expression* (**??**) to allocate `size` bytes of storage. The second form is called for a type with new-extended alignment, and allocates storage with the specified alignment. The first form is called otherwise, and allocates storage suitably aligned to represent any array object of that size or smaller, provided the object's type does not have new-extended alignment.[219]

2 *Replaceable:* A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

3 *Required behavior:* Same as for the corresponding single-object forms. This requirement is binding on any replacement versions of these functions.

4 *Default behavior:* Returns `operator new(size)`, or `operator new(size, alignment)`, respectively.

```
[[nodiscard]] void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment,
                                   const std::nothrow_t&) noexcept;
```

5 *Effects:* Same as above, except that these are called by a placement version of a *new-expression* when a C++ program prefers a null pointer result as an error indication, instead of a `bad_alloc` exception.

6 *Replaceable:* A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

7 *Required behavior:* Return a non-null pointer to suitably aligned storage (**??**), or else return a null pointer. Each of these nothrow versions of `operator new[]` returns a pointer obtained as if acquired from the (possibly replaced) corresponding non-placement function. This requirement is binding on any replacement versions of these functions.

8 *Default behavior:* Calls `operator new[](size)`, or `operator new[](size, alignment)`, respectively. If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

```
void operator delete[](void* ptr) noexcept;
void operator delete[](void* ptr, std::size_t size) noexcept;
void operator delete[](void* ptr, std::align_val_t alignment) noexcept;
void operator delete[](void* ptr, std::size_t size, std::align_val_t alignment) noexcept;
```

9 *Effects:* The deallocation functions (**??**) called by the array form of a *delete-expression* to render the value of `ptr` invalid.

10 *Replaceable:* A C++ program may define functions with any of these function signatures, and thereby displace the default versions defined by the C++ standard library. If a function without a `size` parameter is defined, the program should also define the corresponding function with a `size` parameter. If a function with a `size` parameter is defined, the program shall also define the corresponding version without the `size` parameter. [*Note*: The default behavior below may change in the future, which will require replacing both deallocation functions when replacing the allocation function. — *end note*]

11 ~~*Requires:*~~ *Expects:* `ptr` ~~shall be a null pointer or its value shall represent~~is a null pointer or its value represents the address of a block of memory allocated by an earlier call to a (possibly replaced) `operator new[](std::size_t)` or `operator new[](std::size_t, std::align_val_t)` which has not been invalidated by an intervening call to `operator delete[]`.

12 ~~*Requires:*~~ *Expects:* If an implementation has strict pointer safety (**??**) then `ptr` ~~shall be~~is a safely-derived pointer.

13 ~~*Requires:*~~ *Expects:* If the `alignment` parameter is not present, `ptr` ~~shall have been~~was returned by an allocation function without an `alignment` parameter. If present, the `alignment` argument ~~shall equal~~is equal to the `alignment` argument passed to the allocation function that returned `ptr`. If present, the `size` argument ~~shall equal~~is equal to the `size` argument passed to the allocation function that returned `ptr`.

14 *Required behavior:* A call to an `operator delete[]` with a `size` parameter may be changed to a call to the corresponding `operator delete[]` without a `size` parameter, without affecting memory allocation.

219) It is not the direct responsibility of `operator new[]` or `operator delete[]` to note the repetition count or element size of the array. Those operations are performed elsewhere in the array `new` and `delete` expressions. The array `new` expression, may, however, increase the `size` argument to `operator new[]` to obtain space to store supplemental information.

[*Note*: A conforming implementation is for `operator delete[](void* ptr, std::size_t size)` to simply call `operator delete[](ptr)`. — *end note*]

15 *Default behavior:* The functions that have a `size` parameter forward their other parameters to the corresponding function without a `size` parameter. The functions that do not have a `size` parameter forward their parameters to the corresponding `operator delete` (single-object) function.

```
void operator delete[](void* ptr, const std::nothrow_t&) noexcept;
void operator delete[](void* ptr, std::align_val_t alignment, const std::nothrow_t&) noexcept;
```

16 *Effects:* The deallocation functions (**??**) called by the implementation to render the value of `ptr` invalid when the constructor invoked from a nothrow placement version of the array *new-expression* throws an exception.

17 *Replaceable:* A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

18 ~~*Requires:*~~ *Expects:* `ptr` ~~shall be a null pointer or its value shall represent~~is a null pointer or its value represents the address of a block of memory allocated by an earlier call to a (possibly replaced) `operator new[](std::size_t)` or `operator new[](std::size_t, std::align_val_t)` which has not been invalidated by an intervening call to `operator delete[]`.

19 ~~*Requires:*~~ *Expects:* If an implementation has strict pointer safety (**??**) then `ptr` ~~shall be~~is a safely-derived pointer.

20 ~~*Requires:*~~ *Expects:* If the `alignment` parameter is not present, `ptr` ~~shall have been~~was returned by an allocation function without an `alignment` parameter. If present, the `alignment` argument ~~shall equal~~is equal to the `alignment` argument passed to the allocation function that returned `ptr`.

21 *Default behavior:* Calls `operator delete[](ptr)`, or `operator delete[](ptr, alignment)`, respectively.

16.6.2.3 Non-allocating forms [new.delete.placement]

1 These functions are reserved; a C++ program may not define functions that displace the versions in the C++ standard library (**??**). The provisions of **??** do not apply to these reserved placement forms of `operator new` and `operator delete`.

```
[[nodiscard]] void* operator new(std::size_t size, void* ptr) noexcept;
```

2 *Returns:* `ptr`.

3 *Remarks:* Intentionally performs no other action.

4 [*Example*: This can be useful for constructing an object at a known address:

```
void* place = operator new(sizeof(Something));
Something* p = new (place) Something();
```

— *end example*]

```
[[nodiscard]] void* operator new[](std::size_t size, void* ptr) noexcept;
```

5 *Returns:* `ptr`.

6 *Remarks:* Intentionally performs no other action.

```
void operator delete(void* ptr, void*) noexcept;
```

7 *Effects:* Intentionally performs no action.

8 ~~*Requires:*~~ *Expects:* If an implementation has strict pointer safety (**??**) then `ptr` ~~shall be~~is a safely-derived pointer.

9 *Remarks:* Default function called when any part of the initialization in a placement *new-expression* that invokes the library's non-array placement operator new terminates by throwing an exception (**??**).

```
void operator delete[](void* ptr, void*) noexcept;
```

10 *Effects:* Intentionally performs no action.

11 ~~*Requires:*~~ *Expects:* If an implementation has strict pointer safety (**??**) then `ptr` ~~shall be~~is a safely-derived pointer.

12 *Remarks:* Default function called when any part of the initialization in a placement *new-expression* that invokes the library's array placement operator new terminates by throwing an exception (**??**).

16.6.2.4 Data races [new.delete.dataraces]

1 For purposes of determining the existence of data races, the library versions of `operator new`, user replacement versions of global `operator new`, the C standard library functions `aligned_alloc`, `calloc`, and `malloc`, the library versions of `operator delete`, user replacement versions of `operator delete`, the C standard library function `free`, and the C standard library function `realloc` shall not introduce a data race (**??**). Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before (**??**) the next allocation (if any) in this order.

16.6.3 Storage allocation errors [alloc.errors]

16.6.3.1 Class `bad_alloc` [bad.alloc]

```
namespace std {
  class bad_alloc : public exception {
  public:
    bad_alloc() noexcept;
    bad_alloc(const bad_alloc&) noexcept;
    bad_alloc& operator=(const bad_alloc&) noexcept;
    const char* what() const noexcept override;
  };
}
```

1 The class `bad_alloc` defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage.

```
bad_alloc() noexcept;
```

2 *Effects:* Constructs an object of class `bad_alloc`.

```
bad_alloc(const bad_alloc&) noexcept;
bad_alloc& operator=(const bad_alloc&) noexcept;
```

3 *Effects:* Copies an object of class `bad_alloc`.

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

4 *Returns:* An implementation-defined NTBS.

5 *Remarks:* The message may be a null-terminated multibyte string (**??**), suitable for conversion and display as a `wstring` (**??**, **??**).

16.6.3.2 Class `bad_array_new_length` [new.badlength]

```
namespace std {
  class bad_array_new_length : public bad_alloc {
  public:
    bad_array_new_length() noexcept;
    const char* what() const noexcept override;
  };
}
```

1 The class `bad_array_new_length` defines the type of objects thrown as exceptions by the implementation to report an attempt to allocate an array of size less than zero or greater than an implementation-defined limit (**??**).

```
bad_array_new_length() noexcept;
```

2 *Effects:* Constructs an object of class `bad_array_new_length`.

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

3 *Returns:* An implementation-defined NTBS.

4 *Remarks:* The message may be a null-terminated multibyte string (**??**), suitable for conversion and display as a `wstring` (**??**, **??**).

16.6.3.3 Type `new_handler` [new.handler]

```
using new_handler = void (*)();
```

1 The type of a *handler function* to be called by `operator new()` or `operator new[]()` (16.6.2) when they cannot satisfy a request for additional storage.

2 *Required behavior:* A `new_handler` shall perform one of the following:

(2.1) — make more storage available for allocation and then return;

(2.2) — throw an exception of type `bad_alloc` or a class derived from `bad_alloc`;

(2.3) — terminate execution of the program without returning to the caller.

16.6.3.4 `set_new_handler` [set.new.handler]

```
new_handler set_new_handler(new_handler new_p) noexcept;
```

1 *Effects:* Establishes the function designated by `new_p` as the current `new_handler`.

2 *Returns:* The previous `new_handler`.

3 *Remarks:* The initial `new_handler` is a null pointer.

16.6.3.5 `get_new_handler` [get.new.handler]

```
new_handler get_new_handler() noexcept;
```

1 *Returns:* The current `new_handler`. [*Note*: This may be a null pointer value. — *end note*]

16.6.4 Pointer optimization barrier [ptr.launder]

```
template<class T> [[nodiscard]] constexpr T* launder(T* p) noexcept;
```

1 *Mandates:* `!is_function_v<T> && !is_void_v<T>` is `true`.

2 ~~*Requires:*~~ *Expects:* `p` represents the address A of a byte in memory. An object X that is within its lifetime (**??**) and whose type is similar (**??**) to `T` is located at the address A. All bytes of storage that would be reachable through the result are reachable through `p` (see below).

3 *Returns:* A value of type `T *` that points to `X`.

4 *Remarks:* An invocation of this function may be used in a core constant expression whenever the value of its argument may be used in a core constant expression. A byte of storage is reachable through a pointer value that points to an object Y if it is within the storage occupied by Y, an object that is pointer-interconvertible with Y, or the immediately-enclosing array object if Y is an array element. ~~The program is ill-formed if `T` is a function type or *cv* `void`.~~

5 [*Note*: If a new object is created in storage occupied by an existing object of the same type, a pointer to the original object can be used to refer to the new object unless the type contains `const` or reference members; in the latter cases, this function can be used to obtain a usable pointer to the new object. See **??**. — *end note*]

6 [*Example*:

```
struct X { const int n; };
X *p = new X{3};
const int a = p->n;
new (p) X{5};                       // p does not point to new object (??) because X::n is const
const int b = p->n;                 // undefined behavior
const int c = std::launder(p)->n;   // OK
```

— *end example*]

16.6.5 Hardware interference size [hardware.interference]

```
inline constexpr size_t hardware_destructive_interference_size = implementation-defined;
```

1 This number is the minimum recommended offset between two concurrently-accessed objects to avoid additional performance degradation due to contention introduced by the implementation. It shall be at least `alignof(max_align_t)`.

[*Example*:

```
struct keep_apart {
  alignas(hardware_destructive_interference_size) atomic<int> cat;
  alignas(hardware_destructive_interference_size) atomic<int> dog;
};
```

— *end example*]

```
inline constexpr size_t hardware_constructive_interference_size = implementation-defined;
```

2 This number is the maximum recommended size of contiguous memory occupied by two objects accessed with temporal locality by concurrent threads. It shall be at least `alignof(max_align_t)`.

[*Example*:

```
struct together {
  atomic<int> dog;
  int puppy;
};
struct kennel {
  // Other data members...
  alignas(sizeof(together)) together pack;
  // Other data members...
};
static_assert(sizeof(together) <= hardware_constructive_interference_size);
```

— *end example*]

16.7 Type identification [support.rtti]

1 The header `<typeinfo>` defines a type associated with type information generated by the implementation. It also defines two types for reporting dynamic type identification errors.

16.7.1 Header `<typeinfo>` synopsis [typeinfo.syn]

```
namespace std {
  class type_info;
  class bad_cast;
  class bad_typeid;
}
```

16.7.2 Class `type_info` [type.info]

```
namespace std {
  class type_info {
  public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const noexcept;
    bool operator!=(const type_info& rhs) const noexcept;
    bool before(const type_info& rhs) const noexcept;
    size_t hash_code() const noexcept;
    const char* name() const noexcept;

    type_info(const type_info& rhs) = delete;                    // cannot be copied
    type_info& operator=(const type_info& rhs) = delete;         // cannot be copied
  };
}
```

1 The class `type_info` describes type information generated by the implementation (**??**). Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for equality or collating order. The names, encoding rule, and collating sequence for types are all unspecified and may differ between programs.

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

2 *Effects:* Compares the current object with `rhs`.

3 *Returns:* `true` if the two values describe the same type.

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

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

```
bool before(const type_info& rhs) const noexcept;
```

Effects: Compares the current object with `rhs`.

Returns: `true` if `*this` precedes `rhs` in the implementation's collation order.

```
size_t hash_code() const noexcept;
```

Returns: An unspecified value, except that within a single execution of the program, it shall return the same value for any two `type_info` objects which compare equal.

Remarks: An implementation should return different values for two `type_info` objects which do not compare equal.

```
const char* name() const noexcept;
```

Returns: An implementation-defined NTBS.

Remarks: The message may be a null-terminated multibyte string (**??**), suitable for conversion and display as a `wstring` (**??**, **??**)

16.7.3 Class `bad_cast` [bad.cast]

```
namespace std {
  class bad_cast : public exception {
  public:
    bad_cast() noexcept;
    bad_cast(const bad_cast&) noexcept;
    bad_cast& operator=(const bad_cast&) noexcept;
    const char* what() const noexcept override;
  };
}
```

The class `bad_cast` defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid `dynamic_cast` expression (**??**).

```
bad_cast() noexcept;
```

Effects: Constructs an object of class `bad_cast`.

```
bad_cast(const bad_cast&) noexcept;
bad_cast& operator=(const bad_cast&) noexcept;
```

Effects: Copies an object of class `bad_cast`.

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

Returns: An implementation-defined NTBS.

Remarks: The message may be a null-terminated multibyte string (**??**), suitable for conversion and display as a `wstring` (**??**, **??**)

16.7.4 Class `bad_typeid` [bad.typeid]

```
namespace std {
  class bad_typeid : public exception {
  public:
    bad_typeid() noexcept;
    bad_typeid(const bad_typeid&) noexcept;
    bad_typeid& operator=(const bad_typeid&) noexcept;
    const char* what() const noexcept override;
  };
}
```

The class `bad_typeid` defines the type of objects thrown as exceptions by the implementation to report a null pointer in a `typeid` expression (**??**).

```
bad_typeid() noexcept;
```

Effects: Constructs an object of class `bad_typeid`.

```
bad_typeid(const bad_typeid&) noexcept;
bad_typeid& operator=(const bad_typeid&) noexcept;
```

3 *Effects:* Copies an object of class `bad_typeid`.

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

4 *Returns:* An implementation-defined NTBS.

5 *Remarks:* The message may be a null-terminated multibyte string (**??**), suitable for conversion and display as a `wstring` (**??**, **??**)

16.8 Contract violation handling [support.contract]

16.8.1 Header `<contract>` synopsis [contract.syn]

The header `<contract>` defines a type for reporting information about contract violations generated by the implementation.

```
namespace std {
  class contract_violation;
}
```

16.8.2 Class `contract_violation` [support.contract.cviol]

```
namespace std {
  class contract_violation {
  public:
    uint_least32_t line_number() const noexcept;
    string_view file_name() const noexcept;
    string_view function_name() const noexcept;
    string_view comment() const noexcept;
    string_view assertion_level() const noexcept;
  };
}
```

1 The class `contract_violation` describes information about a contract violation generated by the implementation.

```
uint_least32_t line_number() const noexcept;
```

2 *Returns:* The source code location where the contract violation happened (**??**). If the location is unknown, an implementation may return `0`.

```
string_view file_name() const noexcept;
```

3 *Returns:* The source file name where the contract violation happened (**??**). If the file name is unknown, an implementation may return `string_view{}`.

```
string_view function_name() const noexcept;
```

4 *Returns:* The name of the function where the contract violation happened (**??**). If the function name is unknown, an implementation may return `string_view{}`.

```
string_view comment() const noexcept;
```

5 *Returns:* Implementation-defined text describing the predicate of the violated contract.

```
string_view assertion_level() const noexcept;
```

6 *Returns:* Text describing the *assertion-level* of the violated contract.

16.9 Exception handling [support.exception]

1 The header `<exception>` defines several types and functions related to the handling of exceptions in a C++ program.

16.9.1 Header `<exception>` synopsis [exception.syn]

```
namespace std {
  class exception;
  class bad_exception;
  class nested_exception;
```

```
  using terminate_handler = void (*)();
  terminate_handler get_terminate() noexcept;
  terminate_handler set_terminate(terminate_handler f) noexcept;
  [[noreturn]] void terminate() noexcept;

  int uncaught_exceptions() noexcept;

  using exception_ptr = unspecified;

  exception_ptr current_exception() noexcept;
  [[noreturn]] void rethrow_exception(exception_ptr p);
  template<class E> exception_ptr make_exception_ptr(E e) noexcept;

  template<class T> [[noreturn]] void throw_with_nested(T&& t);
  template<class E> void rethrow_if_nested(const E& e);
}
```

16.9.2 Class `exception` [exception]

```
namespace std {
  class exception {
  public:
    exception() noexcept;
    exception(const exception&) noexcept;
    exception& operator=(const exception&) noexcept;
    virtual ~exception();
    virtual const char* what() const noexcept;
  };
}
```

1 The class `exception` defines the base class for the types of objects thrown as exceptions by C++ standard library components, and certain expressions, to report errors detected during program execution.

2 Each standard library class `T` that derives from class `exception` shall have a publicly accessible copy constructor and a publicly accessible copy assignment operator that do not exit with an exception. These member functions shall meet the following postcondition: If two objects `lhs` and `rhs` both have dynamic type `T` and `lhs` is a copy of `rhs`, then `strcmp(lhs.what(), rhs.what())` shall equal 0.

```
exception() noexcept;
```

3 *Effects:* Constructs an object of class `exception`.

```
exception(const exception& rhs) noexcept;
exception& operator=(const exception& rhs) noexcept;
```

4 ~~*Effects:* Copies an `exception` object.~~

5 *Ensures:* If `*this` and `rhs` both have dynamic type `exception` then the value of the expression `strcmp(what(), rhs.what())` shall equal 0.

```
virtual ~exception();
```

6 *Effects:* Destroys an object of class `exception`.

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

7 *Returns:* An implementation-defined NTBS.

8 *Remarks:* The message may be a null-terminated multibyte string (**??**), suitable for conversion and display as a `wstring` (**??**, **??**). The return value remains valid until the exception object from which it is obtained is destroyed or a non-`const` member function of the exception object is called.

16.9.3 Class `bad_exception` [bad.exception]

```
namespace std {
  class bad_exception : public exception {
  public:
    bad_exception() noexcept;
    bad_exception(const bad_exception&) noexcept;
    bad_exception& operator=(const bad_exception&) noexcept;
```

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

1 The class `bad_exception` defines the type of the object referenced by the `exception_ptr` returned from a call to `current_exception` (16.9.6) when the currently active exception object fails to copy.

```
bad_exception() noexcept;
```

2 *Effects:* Constructs an object of class `bad_exception`.

```
bad_exception(const bad_exception&) noexcept;
bad_exception& operator=(const bad_exception&) noexcept;
```

3 *Effects:* Copies an object of class `bad_exception`.

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

4 *Returns:* An implementation-defined NTBS.

5 *Remarks:* The message may be a null-terminated multibyte string (**??**), suitable for conversion and display as a `wstring` (**??**, **??**).

16.9.4 Abnormal termination [exception.terminate]

16.9.4.1 Type `terminate_handler` [terminate.handler]

```
using terminate_handler = void (*)();
```

1 The type of a *handler function* to be called by `std::terminate()` when terminating exception processing.

2 *Required behavior:* A `terminate_handler` shall terminate execution of the program without returning to the caller.

3 *Default behavior:* The implementation's default `terminate_handler` calls `abort()`.

16.9.4.2 `set_terminate` [set.terminate]

```
terminate_handler set_terminate(terminate_handler f) noexcept;
```

1 *Effects:* Establishes the function designated by `f` as the current handler function for terminating exception processing.

2 *Remarks:* It is unspecified whether a null pointer value designates the default `terminate_handler`.

3 *Returns:* The previous `terminate_handler`.

16.9.4.3 `get_terminate` [get.terminate]

```
terminate_handler get_terminate() noexcept;
```

1 *Returns:* The current `terminate_handler`. [*Note*: This may be a null pointer value. — *end note*]

16.9.4.4 `terminate` [terminate]

```
[[noreturn]] void terminate() noexcept;
```

1 *Remarks:* Called by the implementation when exception handling must be abandoned for any of several reasons (**??**). May also be called directly by the program.

2 *Effects:* Calls a `terminate_handler` function. It is unspecified which `terminate_handler` function will be called if an exception is active during a call to `set_terminate`. Otherwise calls the current `terminate_handler` function. [*Note*: A default `terminate_handler` is always considered a callable handler in this context. — *end note*]

16.9.5 `uncaught_exceptions` [uncaught.exceptions]

```
int uncaught_exceptions() noexcept;
```

1 *Returns:* The number of uncaught exceptions (**??**).

2 *Remarks:* When `uncaught_exceptions() > 0`, throwing an exception can result in a call of the function `std::terminate` (**??**).

16.9.6 Exception propagation [propagation]

```
using exception_ptr = unspecified;
```

1 The type `exception_ptr` can be used to refer to an exception object.

2 `exception_ptr` ~~shall satisfy~~meets the requirements of *Cpp17NullablePointer* (Table **??**).

3 Two non-null values of type `exception_ptr` are equivalent and compare equal if and only if they refer to the same exception.

4 The default constructor of `exception_ptr` produces the null value of the type.

5 `exception_ptr` shall not be implicitly convertible to any arithmetic, enumeration, or pointer type.

6 [*Note*: An implementation might use a reference-counted smart pointer as `exception_ptr`. — *end note*]

7 For purposes of determining the presence of a data race, operations on `exception_ptr` objects shall access and modify only the `exception_ptr` objects themselves and not the exceptions they refer to. Use of `rethrow_exception` on `exception_ptr` objects that refer to the same exception object shall not introduce a data race. [*Note*: If `rethrow_exception` rethrows the same exception object (rather than a copy), concurrent access to that rethrown exception object may introduce a data race. Changes in the number of `exception_ptr` objects that refer to a particular exception do not introduce a data race. — *end note*]

```
exception_ptr current_exception() noexcept;
```

8 *Returns:* An `exception_ptr` object that refers to the currently handled exception (**??**) or a copy of the currently handled exception, or a null `exception_ptr` object if no exception is being handled. The referenced object shall remain valid at least as long as there is an `exception_ptr` object that refers to it. If the function needs to allocate memory and the attempt fails, it returns an `exception_ptr` object that refers to an instance of `bad_alloc`. It is unspecified whether the return values of two successive calls to `current_exception` refer to the same exception object. [*Note*: That is, it is unspecified whether `current_exception` creates a new copy each time it is called. — *end note*] If the attempt to copy the current exception object throws an exception, the function returns an `exception_ptr` object that refers to the thrown exception or, if this is not possible, to an instance of `bad_exception`. [*Note*: The copy constructor of the thrown exception may also fail, so the implementation is allowed to substitute a `bad_exception` object to avoid infinite recursion. — *end note*]

```
[[noreturn]] void rethrow_exception(exception_ptr p);
```

9 ~~*Requires:*~~ *Expects:* `p` ~~shall not be~~is not a null pointer.

10 *Throws:* The exception object to which `p` refers.

```
template<class E> exception_ptr make_exception_ptr(E e) noexcept;
```

11 *Effects:* Creates an `exception_ptr` object that refers to a copy of `e`, as if:

```
try {
  throw e;
} catch(...) {
  return current_exception();
}
```

12 [*Note*: This function is provided for convenience and efficiency reasons. — *end note*]

16.9.7 `nested_exception` [except.nested]

```
namespace std {
  class nested_exception {
  public:
    nested_exception() noexcept;
    nested_exception(const nested_exception&) noexcept = default;
    nested_exception& operator=(const nested_exception&) noexcept = default;
    virtual ~nested_exception() = default;

    // access functions
    [[noreturn]] void rethrow_nested() const;
    exception_ptr nested_ptr() const noexcept;
```

```
  };

  template<class T> [[noreturn]] void throw_with_nested(T&& t);
  template<class E> void rethrow_if_nested(const E& e);
}
```

1 The class `nested_exception` is designed for use as a mixin through multiple inheritance. It captures the currently handled exception and stores it for later use.

2 [*Note*: `nested_exception` has a virtual destructor to make it a polymorphic class. Its presence can be tested for with `dynamic_cast`. — *end note*]

`nested_exception() noexcept;`

3 *Effects:* The constructor calls `current_exception()` and stores the returned value.

`[[noreturn]] void rethrow_nested() const;`

4 *Effects:* If `nested_ptr()` returns a null pointer, the function calls the function `std::terminate`. Otherwise, it throws the stored exception captured by `*this`.

`exception_ptr nested_ptr() const noexcept;`

5 *Returns:* The stored exception captured by this `nested_exception` object.

`template<class T> [[noreturn]] void throw_with_nested(T&& t);`

6 Let `U` be `decay_t<T>`.

7 ~~*Requires:*~~ *Expects:* `U` ~~shall be~~meets the *Cpp17CopyConstructible* requirements.

8 *Throws:* If `is_class_v<U> && !is_final_v<U> && !is_base_of_v<nested_exception, U>` is `true`, an exception of unspecified type that is publicly derived from both `U` and `nested_exception` and constructed from `std::forward<T>(t)`, otherwise `std::forward<T>(t)`.

`template<class E> void rethrow_if_nested(const E& e);`

9 *Effects:* If `E` is not a polymorphic class type, or if `nested_exception` is an inaccessible or ambiguous base class of `E`, there is no effect. Otherwise, performs:

```
if (auto p = dynamic_cast<const nested_exception*>(addressof(e)))
  p->rethrow_nested();
```

16.10 Initializer lists [support.initlist]

1 The header `<initializer_list>` defines a class template and several support functions related to list-initialization (see **??**). All functions specified in this subclause are signal-safe (16.12.4).

16.10.1 Header `<initializer_list>` synopsis [initializer_list.syn]

```
namespace std {
  template<class E> class initializer_list {
  public:
    using value_type      = E;
    using reference       = const E&;
    using const_reference = const E&;
    using size_type       = size_t;

    using iterator        = const E*;
    using const_iterator  = const E*;

    constexpr initializer_list() noexcept;

    constexpr size_t size() const noexcept;      // number of elements
    constexpr const E* begin() const noexcept;   // first element
    constexpr const E* end() const noexcept;     // one past the last element
  };

  // 16.10.4, initializer list range access
  template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;
```

```
  template<class E> constexpr const E* end(initializer_list<E> il) noexcept;
}
```

1 An object of type `initializer_list<E>` provides access to an array of objects of type `const E`. [*Note*: A pair of pointers or a pointer plus a length would be obvious representations for `initializer_list`. `initializer_list` is used to implement initializer lists as specified in **??**. Copying an initializer list does not copy the underlying elements. — *end note*]

2 If an explicit specialization or partial specialization of `initializer_list` is declared, the program is ill-formed.

16.10.2 Initializer list constructors [support.initlist.cons]

```
constexpr initializer_list() noexcept;
```

1 ~~*Effects:* Constructs an empty `initializer_list` object.~~

2 *Ensures:* `size() == 0`.

16.10.3 Initializer list access [support.initlist.access]

```
constexpr const E* begin() const noexcept;
```

1 *Returns:* A pointer to the beginning of the array. If `size() == 0` the values of `begin()` and `end()` are unspecified but they shall be identical.

```
constexpr const E* end() const noexcept;
```

2 *Returns:* `begin() + size()`.

```
constexpr size_t size() const noexcept;
```

3 *Returns:* The number of elements in the array.

4 *Complexity:* Constant time.

16.10.4 Initializer list range access [support.initlist.range]

```
template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;
```

1 *Returns:* `il.begin()`.

```
template<class E> constexpr const E* end(initializer_list<E> il) noexcept;
```

2 *Returns:* `il.end()`.

16.11 Comparisons [cmp]

16.11.1 Header <compare> synopsis [compare.syn]

1 The header `<compare>` specifies types, objects, and functions for use primarily in connection with the three-way comparison operator (**??**).

```
namespace std {
  // 16.11.2, comparison category types
  class weak_equality;
  class strong_equality;
  class partial_ordering;
  class weak_ordering;
  class strong_ordering;

  // named comparison functions
  constexpr bool is_eq  (weak_equality cmp) noexcept    { return cmp == 0; }
  constexpr bool is_neq (weak_equality cmp) noexcept    { return cmp != 0; }
  constexpr bool is_lt  (partial_ordering cmp) noexcept { return cmp < 0; }
  constexpr bool is_lteq(partial_ordering cmp) noexcept { return cmp <= 0; }
  constexpr bool is_gt  (partial_ordering cmp) noexcept { return cmp > 0; }
  constexpr bool is_gteq(partial_ordering cmp) noexcept { return cmp >= 0; }
```

```
  // 16.11.3, common comparison category type
  template<class... Ts>
  struct common_comparison_category {
    using type = see below;
  };
  template<class... Ts>
    using common_comparison_category_t = typename common_comparison_category<Ts...>::type;

  // 16.11.4, comparison algorithms
  template<class T> constexpr strong_ordering strong_order(const T& a, const T& b);
  template<class T> constexpr weak_ordering weak_order(const T& a, const T& b);
  template<class T> constexpr partial_ordering partial_order(const T& a, const T& b);
  template<class T> constexpr strong_equality strong_equal(const T& a, const T& b);
  template<class T> constexpr weak_equality weak_equal(const T& a, const T& b);
}
```

16.11.2 Comparison category types [cmp.categories]

16.11.2.1 Preamble [cmp.categories.pre]

1 The types `weak_equality`, `strong_equality`, `partial_ordering`, `weak_ordering`, and `strong_ordering` are collectively termed the *comparison category types*. Each is specified in terms of an exposition-only data member named `value` whose value typically corresponds to that of an enumerator from one of the following exposition-only enumerations:

```
enum class eq { equal = 0, equivalent = equal,
                nonequal = 1, nonequivalent = nonequal };    // exposition only
enum class ord { less = -1, greater = 1 };                   // exposition only
enum class ncmp { unordered = -127 };                        // exposition only
```

2 [*Note*: The types `strong_ordering` and `weak_equality` correspond, respectively, to the terms total ordering and equivalence in mathematics. — *end note*]

3 The relational and equality operators for the comparison category types are specified with an anonymous parameter of unspecified type. This type shall be selected by the implementation such that these parameters can accept literal `0` as a corresponding argument. [*Example*: `nullptr_t` satisfies this requirement. — *end example*] In this context, the behavior of a program that supplies an argument other than a literal `0` is undefined.

4 For the purposes of this subclause, *substitutability* is the property that `f(a) == f(b)` is `true` whenever `a == b` is true, where `f` denotes a function that reads only comparison-salient state that is accessible via the argument's public const members.

16.11.2.2 Class `weak_equality` [cmp.weakeq]

1 The `weak_equality` type is typically used as the result type of a three-way comparison operator (**??**) that (a) admits only equality and inequality comparisons, and (b) does not imply substitutability.

```
namespace std {
  class weak_equality {
    int value;  // exposition only

    // exposition-only constructor
    constexpr explicit weak_equality(eq v) noexcept : value(int(v)) {}  // exposition only

  public:
    // valid values
    static const weak_equality equivalent;
    static const weak_equality nonequivalent;

    // comparisons
    friend constexpr bool operator==(weak_equality v, unspecified) noexcept;
    friend constexpr bool operator!=(weak_equality v, unspecified) noexcept;
    friend constexpr bool operator==(unspecified, weak_equality v) noexcept;
    friend constexpr bool operator!=(unspecified, weak_equality v) noexcept;
    friend constexpr weak_equality operator<=>(weak_equality v, unspecified) noexcept;
    friend constexpr weak_equality operator<=>(unspecified, weak_equality v) noexcept;
  };
```

```
    // valid values' definitions
    inline constexpr weak_equality weak_equality::equivalent(eq::equivalent);
    inline constexpr weak_equality weak_equality::nonequivalent(eq::nonequivalent);
  }

constexpr bool operator==(weak_equality v, unspecified) noexcept;
constexpr bool operator==(unspecified, weak_equality v) noexcept;
```

2 *Returns:* `v.value == 0`.

```
constexpr bool operator!=(weak_equality v, unspecified) noexcept;
constexpr bool operator!=(unspecified, weak_equality v) noexcept;
```

3 *Returns:* `v.value != 0`.

```
constexpr weak_equality operator<=>(weak_equality v, unspecified) noexcept;
constexpr weak_equality operator<=>(unspecified, weak_equality v) noexcept;
```

4 *Returns:* `v`.

16.11.2.3 Class `strong_equality` [cmp.strongeq]

1 The `strong_equality` type is typically used as the result type of a three-way comparison operator (**??**) that (a) admits only equality and inequality comparisons, and (b) does imply substitutability.

```
  namespace std {
    class strong_equality {
      int value;  // exposition only

      // exposition-only constructor
      constexpr explicit strong_equality(eq v) noexcept : value(int(v)) {}    // exposition only

    public:
      // valid values
      static const strong_equality equal;
      static const strong_equality nonequal;
      static const strong_equality equivalent;
      static const strong_equality nonequivalent;

      // conversion
      constexpr operator weak_equality() const noexcept;

      // comparisons
      friend constexpr bool operator==(strong_equality v, unspecified) noexcept;
      friend constexpr bool operator!=(strong_equality v, unspecified) noexcept;
      friend constexpr bool operator==(unspecified, strong_equality v) noexcept;
      friend constexpr bool operator!=(unspecified, strong_equality v) noexcept;
      friend constexpr strong_equality operator<=>(strong_equality v, unspecified) noexcept;
      friend constexpr strong_equality operator<=>(unspecified, strong_equality v) noexcept;
    };

    // valid values' definitions
    inline constexpr strong_equality strong_equality::equal(eq::equal);
    inline constexpr strong_equality strong_equality::nonequal(eq::nonequal);
    inline constexpr strong_equality strong_equality::equivalent(eq::equivalent);
    inline constexpr strong_equality strong_equality::nonequivalent(eq::nonequivalent);
  }

constexpr operator weak_equality() const noexcept;
```

2 *Returns:* `value == 0 ? weak_equality::equivalent : weak_equality::nonequivalent`.

```
constexpr bool operator==(strong_equality v, unspecified) noexcept;
constexpr bool operator==(unspecified, strong_equality v) noexcept;
```

3 *Returns:* `v.value == 0`.

```
constexpr bool operator!=(strong_equality v, unspecified) noexcept;
```

```
constexpr bool operator!=(unspecified, strong_equality v) noexcept;
```

4 *Returns:* `v.value != 0`.

```
constexpr strong_equality operator<=>(strong_equality v, unspecified) noexcept;
constexpr strong_equality operator<=>(unspecified, strong_equality v) noexcept;
```

5 *Returns:* `v`.

16.11.2.4 Class `partial_ordering` [cmp.partialord]

1 The `partial_ordering` type is typically used as the result type of a three-way comparison operator (**??**) that (a) admits all of the six two-way comparison operators (**??**, **??**), (b) does not imply substitutability, and (c) permits two values to be incomparable.[220]

```
namespace std {
  class partial_ordering {
    int value;          // exposition only
    bool is_ordered;    // exposition only

    // exposition-only constructors
    constexpr explicit
      partial_ordering(eq v) noexcept : value(int(v)), is_ordered(true) {}      // exposition only
    constexpr explicit
      partial_ordering(ord v) noexcept : value(int(v)), is_ordered(true) {}     // exposition only
    constexpr explicit
      partial_ordering(ncmp v) noexcept : value(int(v)), is_ordered(false) {}   // exposition only

  public:
    // valid values
    static const partial_ordering less;
    static const partial_ordering equivalent;
    static const partial_ordering greater;
    static const partial_ordering unordered;

    // conversion
    constexpr operator weak_equality() const noexcept;

    // comparisons
    friend constexpr bool operator==(partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator!=(partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator< (partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator> (partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator<=(partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator>=(partial_ordering v, unspecified) noexcept;
    friend constexpr bool operator==(unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator!=(unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator< (unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator> (unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator<=(unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator>=(unspecified, partial_ordering v) noexcept;
    friend constexpr partial_ordering operator<=>(partial_ordering v, unspecified) noexcept;
    friend constexpr partial_ordering operator<=>(unspecified, partial_ordering v) noexcept;
  };

  // valid values' definitions
  inline constexpr partial_ordering partial_ordering::less(ord::less);
  inline constexpr partial_ordering partial_ordering::equivalent(eq::equivalent);
  inline constexpr partial_ordering partial_ordering::greater(ord::greater);
  inline constexpr partial_ordering partial_ordering::unordered(ncmp::unordered);
}
```

220) That is, `a < b`, `a == b`, and `a > b` might all be `false`.

```
constexpr operator weak_equality() const noexcept;
```

2 *Returns:* `value == 0 ? weak_equality::equivalent : weak_equality::nonequivalent`. [*Note*: The result is independent of the `is_ordered` member. — *end note*]

```
constexpr bool operator==(partial_ordering v, unspecified) noexcept;
constexpr bool operator< (partial_ordering v, unspecified) noexcept;
constexpr bool operator> (partial_ordering v, unspecified) noexcept;
constexpr bool operator<=(partial_ordering v, unspecified) noexcept;
constexpr bool operator>=(partial_ordering v, unspecified) noexcept;
```

3 *Returns:* For `operator@`, `v.is_ordered && v.value @ 0`.

```
constexpr bool operator==(unspecified, partial_ordering v) noexcept;
constexpr bool operator< (unspecified, partial_ordering v) noexcept;
constexpr bool operator> (unspecified, partial_ordering v) noexcept;
constexpr bool operator<=(unspecified, partial_ordering v) noexcept;
constexpr bool operator>=(unspecified, partial_ordering v) noexcept;
```

4 *Returns:* For `operator@`, `v.is_ordered && 0 @ v.value`.

```
constexpr bool operator!=(partial_ordering v, unspecified) noexcept;
constexpr bool operator!=(unspecified, partial_ordering v) noexcept;
```

5 *Returns:* For `operator@`, `!v.is_ordered || v.value != 0`.

```
constexpr partial_ordering operator<=>(partial_ordering v, unspecified) noexcept;
```

6 *Returns:* `v`.

```
constexpr partial_ordering operator<=>(unspecified, partial_ordering v) noexcept;
```

7 *Returns:* `v < 0 ? partial_ordering::greater : v > 0 ? partial_ordering::less : v`.

16.11.2.5 Class `weak_ordering` [cmp.weakord]

1 The `weak_ordering` type is typically used as the result type of a three-way comparison operator (**??**) that (a) admits all of the six two-way comparison operators (**??**, **??**), and (b) does not imply substitutability.

```
namespace std {
  class weak_ordering {
    int value;  // exposition only

    // exposition-only constructors
    constexpr explicit weak_ordering(eq v) noexcept : value(int(v)) {}  // exposition only
    constexpr explicit weak_ordering(ord v) noexcept : value(int(v)) {} // exposition only

  public:
    // valid values
    static const weak_ordering less;
    static const weak_ordering equivalent;
    static const weak_ordering greater;

    // conversions
    constexpr operator weak_equality() const noexcept;
    constexpr operator partial_ordering() const noexcept;

    // comparisons
    friend constexpr bool operator==(weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator!=(weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator< (weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator> (weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator<=(weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator>=(weak_ordering v, unspecified) noexcept;
    friend constexpr bool operator==(unspecified, weak_ordering v) noexcept;
    friend constexpr bool operator!=(unspecified, weak_ordering v) noexcept;
    friend constexpr bool operator< (unspecified, weak_ordering v) noexcept;
    friend constexpr bool operator> (unspecified, weak_ordering v) noexcept;
    friend constexpr bool operator<=(unspecified, weak_ordering v) noexcept;
```

```
    friend constexpr bool operator>=(unspecified, weak_ordering v) noexcept;
    friend constexpr weak_ordering operator<=>(weak_ordering v, unspecified) noexcept;
    friend constexpr weak_ordering operator<=>(unspecified, weak_ordering v) noexcept;
  };

  // valid values' definitions
  inline constexpr weak_ordering weak_ordering::less(ord::less);
  inline constexpr weak_ordering weak_ordering::equivalent(eq::equivalent);
  inline constexpr weak_ordering weak_ordering::greater(ord::greater);
}
```

```
constexpr operator weak_equality() const noexcept;
```

2 *Returns:* `value == 0 ? weak_equality::equivalent : weak_equality::nonequivalent`.

```
constexpr operator partial_ordering() const noexcept;
```

3 *Returns:*

```
value == 0 ? partial_ordering::equivalent :
value < 0  ? partial_ordering::less :
             partial_ordering::greater
```

```
constexpr bool operator==(weak_ordering v, unspecified) noexcept;
constexpr bool operator!=(weak_ordering v, unspecified) noexcept;
constexpr bool operator< (weak_ordering v, unspecified) noexcept;
constexpr bool operator> (weak_ordering v, unspecified) noexcept;
constexpr bool operator<=(weak_ordering v, unspecified) noexcept;
constexpr bool operator>=(weak_ordering v, unspecified) noexcept;
```

4 *Returns:* `v.value @ 0` for `operator@`.

```
constexpr bool operator==(unspecified, weak_ordering v) noexcept;
constexpr bool operator!=(unspecified, weak_ordering v) noexcept;
constexpr bool operator< (unspecified, weak_ordering v) noexcept;
constexpr bool operator> (unspecified, weak_ordering v) noexcept;
constexpr bool operator<=(unspecified, weak_ordering v) noexcept;
constexpr bool operator>=(unspecified, weak_ordering v) noexcept;
```

5 *Returns:* `0 @ v.value` for `operator@`.

```
constexpr weak_ordering operator<=>(weak_ordering v, unspecified) noexcept;
```

6 *Returns:* `v`.

```
constexpr weak_ordering operator<=>(unspecified, weak_ordering v) noexcept;
```

7 *Returns:* `v < 0 ? weak_ordering::greater : v > 0 ? weak_ordering::less : v`.

16.11.2.6 Class `strong_ordering` [cmp.strongord]

1 The `strong_ordering` type is typically used as the result type of a three-way comparison operator (**??**) that (a) admits all of the six two-way comparison operators (**??**, **??**), and (b) does imply substitutability.

```
namespace std {
  class strong_ordering {
    int value;  // exposition only

    // exposition-only constructors
    constexpr explicit strong_ordering(eq v) noexcept : value(int(v)) {}    // exposition only
    constexpr explicit strong_ordering(ord v) noexcept : value(int(v)) {}   // exposition only

  public:
    // valid values
    static const strong_ordering less;
    static const strong_ordering equal;
    static const strong_ordering equivalent;
    static const strong_ordering greater;
```

```
    // conversions
    constexpr operator weak_equality() const noexcept;
    constexpr operator strong_equality() const noexcept;
    constexpr operator partial_ordering() const noexcept;
    constexpr operator weak_ordering() const noexcept;

    // comparisons
    friend constexpr bool operator==(strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator!=(strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator< (strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator> (strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator<=(strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator>=(strong_ordering v, unspecified) noexcept;
    friend constexpr bool operator==(unspecified, strong_ordering v) noexcept;
    friend constexpr bool operator!=(unspecified, strong_ordering v) noexcept;
    friend constexpr bool operator< (unspecified, strong_ordering v) noexcept;
    friend constexpr bool operator> (unspecified, strong_ordering v) noexcept;
    friend constexpr bool operator<=(unspecified, strong_ordering v) noexcept;
    friend constexpr bool operator>=(unspecified, strong_ordering v) noexcept;
    friend constexpr strong_ordering operator<=>(strong_ordering v, unspecified) noexcept;
    friend constexpr strong_ordering operator<=>(unspecified, strong_ordering v) noexcept;
  };

  // valid values' definitions
  inline constexpr strong_ordering strong_ordering::less(ord::less);
  inline constexpr strong_ordering strong_ordering::equal(eq::equal);
  inline constexpr strong_ordering strong_ordering::equivalent(eq::equivalent);
  inline constexpr strong_ordering strong_ordering::greater(ord::greater);
}
```

```
constexpr operator weak_equality() const noexcept;
```

2 *Returns:* `value == 0 ? weak_equality::equivalent : weak_equality::nonequivalent`.

```
constexpr operator strong_equality() const noexcept;
```

3 *Returns:* `value == 0 ? strong_equality::equal : strong_equality::nonequal`.

```
constexpr operator partial_ordering() const noexcept;
```

4 *Returns:*

```
value == 0 ? partial_ordering::equivalent :
value < 0  ? partial_ordering::less :
             partial_ordering::greater
```

```
constexpr operator weak_ordering() const noexcept;
```

5 *Returns:*

```
value == 0 ? weak_ordering::equivalent :
value < 0  ? weak_ordering::less :
             weak_ordering::greater
```

```
constexpr bool operator==(strong_ordering v, unspecified) noexcept;
constexpr bool operator!=(strong_ordering v, unspecified) noexcept;
constexpr bool operator< (strong_ordering v, unspecified) noexcept;
constexpr bool operator> (strong_ordering v, unspecified) noexcept;
constexpr bool operator<=(strong_ordering v, unspecified) noexcept;
constexpr bool operator>=(strong_ordering v, unspecified) noexcept;
```

6 *Returns:* `v.value @ 0` for `operator@`.

```
constexpr bool operator==(unspecified, strong_ordering v) noexcept;
constexpr bool operator!=(unspecified, strong_ordering v) noexcept;
constexpr bool operator< (unspecified, strong_ordering v) noexcept;
constexpr bool operator> (unspecified, strong_ordering v) noexcept;
constexpr bool operator<=(unspecified, strong_ordering v) noexcept;
```

```
constexpr bool operator>=(unspecified, strong_ordering v) noexcept;
```

7 *Returns:* `0 @ v.value` for `operator@`.

```
constexpr strong_ordering operator<=>(strong_ordering v, unspecified) noexcept;
```

8 *Returns:* `v`.

```
constexpr strong_ordering operator<=>(unspecified, strong_ordering v) noexcept;
```

9 *Returns:* `v < 0 ? strong_ordering::greater : v > 0 ? strong_ordering::less : v`.

16.11.3 Class template `common_comparison_category` [cmp.common]

1 The type `common_comparison_category` provides an alias for the strongest comparison category to which all of the template arguments can be converted. [*Note*: A comparison category type is stronger than another if they are distinct types and an instance of the former can be converted to an instance of the latter. — *end note*]

```
template<class... Ts>
struct common_comparison_category {
  using type = see below;
};
```

2 *Remarks:* The member *typedef-name* `type` denotes the common comparison type (**??**) of `Ts...`, the expanded parameter pack. [*Note*: This is well-defined even if the expansion is empty or includes a type that is not a comparison category type. — *end note*]

16.11.4 Comparison algorithms [cmp.alg]

```
template<class T> constexpr strong_ordering strong_order(const T& a, const T& b);
```

1 *Effects:* Compares two values and produces a result of type `strong_ordering`:

(1.1) — If `numeric_limits<T>::is_iec559` is `true`, returns a result of type `strong_ordering` that is consistent with the `totalOrder` operation as specified in ISO/IEC/IEEE 60559.

(1.2) — Otherwise, returns `a <=> b` if that expression is well-formed and convertible to `strong_ordering`.

(1.3) — Otherwise, if the expression `a <=> b` is well-formed, then the function is defined as deleted.

(1.4) — Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to `bool`, then

(1.4.1) — if `a == b` is `true`, returns `strong_ordering::equal`;

(1.4.2) — otherwise, if `a < b` is `true`, returns `strong_ordering::less`;

(1.4.3) — otherwise, returns `strong_ordering::greater`.

(1.5) — Otherwise, the function is defined as deleted.

```
template<class T> constexpr weak_ordering weak_order(const T& a, const T& b);
```

2 *Effects:* Compares two values and produces a result of type `weak_ordering`:

(2.1) — Returns `a <=> b` if that expression is well-formed and convertible to `weak_ordering`.

(2.2) — Otherwise, if the expression `a <=> b` is well-formed, then the function is defined as deleted.

(2.3) — Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to `bool`, then

(2.3.1) — if `a == b` is `true`, returns `weak_ordering::equivalent`;

(2.3.2) — otherwise, if `a < b` is `true`, returns `weak_ordering::less`;

(2.3.3) — otherwise, returns `weak_ordering::greater`.

(2.4) — Otherwise, the function is defined as deleted.

```
template<class T> constexpr partial_ordering partial_order(const T& a, const T& b);
```

3 *Effects:* Compares two values and produces a result of type `partial_ordering`:

(3.1) — Returns `a <=> b` if that expression is well-formed and convertible to `partial_ordering`.

(3.2) — Otherwise, if the expression `a <=> b` is well-formed, then the function is defined as deleted.

(3.3) — Otherwise, if the expressions `a == b` and `a < b` are each well-formed and convertible to `bool`, then

(3.3.1) — if `a == b` is `true`, returns `partial_ordering::equivalent`;

(3.3.2) — otherwise, if `a < b` is `true`, returns `partial_ordering::less`;

(3.3.3) — otherwise, returns `partial_ordering::greater`.

(3.4) — Otherwise, the function is defined as deleted.

```
template<class T> constexpr strong_equality strong_equal(const T& a, const T& b);
```

4 *Effects:* Compares two values and produces a result of type `strong_equality`:

(4.1) — Returns `a <=> b` if that expression is well-formed and convertible to `strong_equality`.

(4.2) — Otherwise, if the expression `a <=> b` is well-formed, then the function is defined as deleted.

(4.3) — Otherwise, if the expression `a == b` is well-formed and convertible to `bool`, then

(4.3.1) — if `a == b` is `true`, returns `strong_equality::equal`;

(4.3.2) — otherwise, returns `strong_equality::nonequal`.

(4.4) — Otherwise, the function is defined as deleted.

```
template<class T> constexpr weak_equality weak_equal(const T& a, const T& b);
```

5 *Effects:* Compares two values and produces a result of type `weak_equality`:

(5.1) — Returns `a <=> b` if that expression is well-formed and convertible to `weak_equality`.

(5.2) — Otherwise, if the expression `a <=> b` is well-formed, then the function is defined as deleted.

(5.3) — Otherwise, if the expression `a == b` is well-formed and convertible to `bool`, then

(5.3.1) — if `a == b` is `true`, returns `weak_equality::equivalent`;

(5.3.2) — otherwise, returns `weak_equality::nonequivalent`.

(5.4) — Otherwise, the function is defined as deleted.

16.12 Other runtime support [support.runtime]

1 Headers `<csetjmp>` (nonlocal jumps), `<csignal>` (signal handling), `<cstdarg>` (variable arguments), and `<cstdlib>` (runtime environment `getenv`, `system`), provide further compatibility with C code.

2 Calls to the function `getenv` (16.2.2) shall not introduce a data race (**??**) provided that nothing modifies the environment. [*Note*: Calls to the POSIX functions `setenv` and `putenv` modify the environment. — *end note*]

3 A call to the `setlocale` function (**??**) may introduce a data race with other calls to the `setlocale` function or with calls to functions that are affected by the current C locale. The implementation shall behave as if no library function other than `locale::global` calls the `setlocale` function.

16.12.1 Header <cstdarg> synopsis [cstdarg.syn]

```
namespace std {
  using va_list = see below;
}

#define va_arg(V, P) see below
#define va_copy(VDST, VSRC) see below
#define va_end(V) see below
#define va_start(V, P) see below
```

1 The contents of the header `<cstdarg>` are the same as the C standard library header `<stdarg.h>`, with the following changes: The restrictions that ISO C places on the second parameter to the `va_start` macro in header `<stdarg.h>` are different in this document. The parameter `parmN` is the rightmost parameter in the variable parameter list of the function definition (the one just before the `...`).[221] If the parameter `parmN` is a pack expansion (**??**) or an entity resulting from a lambda capture (**??**), the program is ill-formed, no diagnostic required. If the parameter `parmN` is of a reference type, or of a type that is not compatible with the type that results when passing an argument for which there is no parameter, the behavior is undefined.

221) Note that `va_start` is required to work as specified even if unary `operator&` is overloaded for the type of `parmN`.

See also: ISO C 7.16.1.1

16.12.2 Header <csetjmp> synopsis [csetjmp.syn]

```
namespace std {
  using jmp_buf = see below;
  [[noreturn]] void longjmp(jmp_buf env, int val);
}

#define setjmp(env) see below
```

1 The contents of the header `<csetjmp>` are the same as the C standard library header `<setjmp.h>`.

2 The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this document. A `setjmp`/`longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by `catch` and `throw` would invoke any non-trivial destructors for any automatic objects.

See also: ISO C 7.13

16.12.3 Header <csignal> synopsis [csignal.syn]

```
namespace std {
  using sig_atomic_t = see below;

  // 16.12.4, signal handlers
  extern "C" using signal-handler = void(int);  // exposition only
  signal-handler* signal(int sig, signal-handler* func);

  int raise(int sig);
}

#define SIG_DFL see below
#define SIG_ERR see below
#define SIG_IGN see below
#define SIGABRT see below
#define SIGFPE see below
#define SIGILL see below
#define SIGINT see below
#define SIGSEGV see below
#define SIGTERM see below
```

1 The contents of the header `<csignal>` are the same as the C standard library header `<signal.h>`.

16.12.4 Signal handlers [support.signal]

1 A call to the function `signal` synchronizes with any resulting invocation of the signal handler so installed.

2 A *plain lock-free atomic operation* is an invocation of a function `f` from **??**, such that:

(2.1) — `f` is the function `atomic_is_lock_free()`, or

(2.2) — `f` is the member function `is_lock_free()`, or

(2.3) — `f` is a non-static member function invoked on an object `A`, such that `A.is_lock_free()` yields `true`, or

(2.4) — `f` is a non-member function, and for every pointer-to-atomic argument `A` passed to `f`, `atomic_is_lock_free(A)` yields `true`.

3 An evaluation is *signal-safe* unless it includes one of the following:

(3.1) — a call to any standard library function, except for plain lock-free atomic operations and functions explicitly identified as signal-safe. [*Note*: This implicitly excludes the use of `new` and `delete` expressions that rely on a library-provided memory allocator. — *end note*]

(3.2) — an access to an object with thread storage duration;

(3.3) — a `dynamic_cast` expression;

(3.4) — throwing of an exception;

(3.5) — control entering a *try-block* or *function-try-block*;

(3.6) — initialization of a variable with static storage duration requiring dynamic initialization (**??**, **??**)[222]; or

(3.7) — waiting for the completion of the initialization of a variable with static storage duration (**??**).

A signal handler invocation has undefined behavior if it includes an evaluation that is not signal-safe.

4 The function `signal` is signal-safe if it is invoked with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler.

See also: ISO C 7.14

222) Such initialization might occur because it is the first odr-use (**??**) of that variable.