```
double&& rrd3 = i3;             // rrd3 refers to temporary with value 2.0
```

— *end example*]

In all cases except the last (i.e., implicitly converting the initializer expression to the underlying type of the reference), the reference is said to *bind directly* to the initializer expression.

6 [*Note:* 12.2 describes the lifetime of temporaries bound to references. — *end note*]

8.6.4 List-initialization [dcl.init.list]

1 *List-initialization* is initialization of an object or reference from a *braced-init-list*. Such an initializer is called an *initializer list*, and the comma-separated *initializer-clause*s of the list are called the *elements* of the initializer list. An initializer list may be empty. List-initialization can occur in direct-initialization or copy-initialization contexts; list-initialization in a direct-initialization context is called *direct-list-initialization* and list-initialization in a copy-initialization context is called *copy-list-initialization*. [*Note:* List-initialization can be used

(1.1) — as the initializer in a variable definition (8.6)

(1.2) — as the initializer in a *new-expression* (5.3.4)

(1.3) — in a return statement (6.6.3)

(1.4) — as a *for-range-initializer* (6.5)

(1.5) — as a function argument (5.2.2)

(1.6) — as a subscript (5.2.1)

(1.7) — as an argument to a constructor invocation (8.6, 5.2.3)

(1.8) — as an initializer for a non-static data member (9.2)

(1.9) — in a *mem-initializer* (12.6.2)

(1.10) — on the right-hand side of an assignment (5.18)

[*Example:*

```
int a = {1};
std::complex<double> z{1,2};
new std::vector<std::string>{"once", "upon", "a", "time"};  // 4 string elements
f( {"Nicholas","Annemarie"} );  // pass list of two elements
return { "Norah" };             // return list of one element
int* e {};                      // initialization to zero / null pointer
x = double{1};                  // explicitly construct a double
std::map<std::string,int> anim = { {"bear",4}, {"cassowary",2}, {"tiger",7} };
```

— *end example*] — *end note*]

2 A constructor is an *initializer-list constructor* if its first parameter is of type `std::initializer_list<E>` or reference to possibly cv-qualified `std::initializer_list<E>` for some type `E`, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [*Note:* Initializer-list constructors are favored over other constructors in list-initialization (13.3.1.7). Passing an initializer list as the argument to the constructor template `template<class T> C(T)` of a class `C` does not create an initializer-list constructor, because an initializer list argument causes the corresponding parameter to be a non-deduced context (14.8.2.1). — *end note*] The template `std::initializer_list` is not predefined; if the header `<initializer_list>` is not included prior to a use of `std::initializer_list` — even an implicit use in which the type is not named (7.1.7.4) — the program is ill-formed.

3 List-initialization of an object or reference of type `T` is defined as follows:

(3.1) — If `T` is an aggregate class and the initializer list has a single element of type *cv* `U`, where `U` is `T` or a

©ISO/IEC Dxxxx

```
double&& rrd3 = i3;             // rrd3 refers to temporary with value 2.0
```

— *end example*]

In all cases except the last (i.e., implicitly converting the initializer expression to the underlying type of the reference), the reference is said to *bind directly* to the initializer expression.

6 [*Note:* 12.2 describes the lifetime of temporaries bound to references. — *end note*]

8.6.4 List-initialization [dcl.init.list]

1 *List-initialization* is initialization of an object or reference from a *braced-init-list*. Such an initializer is called an *initializer list*, and the comma-separated *initializer-clause*s of the list are called the *elements* of the initializer list. An initializer list may be empty. List-initialization can occur in direct-initialization or copy-initialization contexts; list-initialization in a direct-initialization context is called *direct-list-initialization* and list-initialization in a copy-initialization context is called *copy-list-initialization*. [*Note:* List-initialization can be used

(1.1) — as the initializer in a variable definition (8.6)

(1.2) — as the initializer in a *new-expression* (5.3.4)

(1.3) — in a return statement (6.6.3)

(1.4) — as a *for-range-initializer* (6.5)

(1.5) — as a function argument (5.2.2)

(1.6) — as a subscript (5.2.1)

(1.7) — as an argument to a constructor invocation (8.6, 5.2.3)

(1.8) — as an initializer for a non-static data member (9.2)

(1.9) — in a *mem-initializer* (12.6.2)

(1.10) — on the right-hand side of an assignment (5.18)

[*Example:*

```
int a = {1};
std::complex<double> z{1,2};
new std::vector<std::string>{"once", "upon", "a", "time"};  // 4 string elements
f( {"Nicholas","Annemarie"} );  // pass list of two elements
return { "Norah" };             // return list of one element
int* e {};                      // initialization to zero / null pointer
x = double{1};                  // explicitly construct a double
std::map<std::string,int> anim = { {"bear",4}, {"cassowary",2}, {"tiger",7} };
```

— *end example*] — *end note*]

2 A constructor is an *initializer-list constructor* if its first parameter is of type `std::initializer_list<E>` or reference to possibly cv-qualified `std::initializer_list<E>` for some type `E`, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [*Note:* Initializer-list constructors are favored over other constructors in list-initialization (13.3.1.7). Passing an initializer list as the argument to the constructor template `template<class T> C(T)` of a class `C` does not create an initializer-list constructor, because an initializer list argument causes the corresponding parameter to be a non-deduced context (14.8.2.1). — *end note*] The template `std::initializer_list` is not predefined; if the header `<initializer_list>` is not included prior to a use of `std::initializer_list` — even an implicit use in which the type is not named (7.1.7.4) — the program is ill-formed.

3 List-initialization of an object or reference of type `T` is defined as follows:

(3.1) — If `T` is an aggregate class and the initializer list has a single element of type *cv* `U`, where `U` is `T` or a

§ 8.6.4 226

assuming that the implementation can construct an `initializer_list` object with a pair of pointers. — *end example*]

6 The array has the same lifetime as any other temporary object (12.2), except that initializing an `initializer_list` object from the array extends the lifetime of the array exactly like binding a reference to a temporary. [*Example:*

```
typedef std::complex<double> cmplx;
std::vector<cmplx> v1 = { 1, 2, 3 };

void f() {
  std::vector<cmplx> v2{ 1, 2, 3 };
  std::initializer_list<int> i3 = { 1, 2, 3 };
}

struct A {
  std::initializer_list<int> i4;
  A() : i4{ 1, 2, 3 } {}  // ill-formed, would create a dangling reference
};
```

For `v1` and `v2`, the `initializer_list` object is a parameter in a function call, so the array created for `{ 1, 2, 3 }` has full-expression lifetime. For `i3`, the `initializer_list` object is a variable, so the array persists for the lifetime of the variable. For `i4`, the `initializer_list` object is initialized in the constructor's *ctor-initializer* as if by binding a temporary array to a reference member, so the program is ill-formed (12.6.2). — *end example*] [*Note:* The implementation is free to allocate the array in read-only memory if an explicit array with the same initializer could be so allocated. — *end note*]

7 A *narrowing conversion* is an implicit conversion

(7.1) — from a floating-point type to an integer type, or

(7.2) — from `long double` to `double` or `float`, or from `double` to `float`, except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly), or

(7.3) — from an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or

(7.4) — from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression whose value after integral promotions will fit into the target type.

[*Note:* As indicated above, such conversions are not allowed at the top level in list-initializations. — *end note*] [*Example:*

```
int x = 999;              // x is not a constant expression
const int y = 999;
const int z = 99;
char c1 = x;              // OK, though it might narrow (in this case, it does narrow)
char c2{x};               // error: might narrow
char c3{y};               // error: narrows (assuming char is 8 bits)
char c4{z};               // OK: no narrowing needed
unsigned char uc1 = {5};  // OK: no narrowing needed
unsigned char uc2 = {-1}; // error: narrows
unsigned int ui1 = {-1};  // error: narrows
signed int si1 =
  { (unsigned int)-1 };   // error: narrows
int ii = {2.0};           // error: narrows
```


assuming that the implementation can construct an `initializer_list` object with a pair of pointers. — *end example*]

6 The array has the same lifetime as any other temporary object (12.2), except that initializing an `initializer_list` object from the array extends the lifetime of the array exactly like binding a reference to a temporary. [*Example:*

```
typedef std::complex<double> cmplx;
std::vector<cmplx> v1 = { 1, 2, 3 };

void f() {
  std::vector<cmplx> v2{ 1, 2, 3 };
  std::initializer_list<int> i3 = { 1, 2, 3 };
}

struct A {
  std::initializer_list<int> i4;
  A() : i4{ 1, 2, 3 } {}  // ill-formed, would create a dangling reference
};
```

For `v1` and `v2`, the `initializer_list` object is a parameter in a function call, so the array created for `{ 1, 2, 3 }` has full-expression lifetime. For `i3`, the `initializer_list` object is a variable, so the array persists for the lifetime of the variable. For `i4`, the `initializer_list` object is initialized in the constructor's *ctor-initializer* as if by binding a temporary array to a reference member, so the program is ill-formed (12.6.2). — *end example*] [*Note:* The implementation is free to allocate the array in read-only memory if an explicit array with the same initializer could be so allocated. — *end note*]

7 A *narrowing conversion* is an implicit conversion

(7.1) — from a floating-point type to an integer type, or

(7.2) — from `long double` to `double` or `float`, or from `double` to `float`, except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly), or

(7.3) — from an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or

(7.4) — from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression whose value after integral promotions will fit into the target type.

[*Note:* As indicated above, such conversions are not allowed at the top level in list-initializations. — *end note*] [*Example:*

```
int x = 999;              // x is not a constant expression
const int y = 999;
const int z = 99;
char c1 = x;              // OK, though it might narrow (in this case, it does narrow)
char c2{x};               // error: might narrow
char c3{y};               // error: narrows (assuming char is 8 bits)
char c4{z};               // OK: no narrowing needed
unsigned char uc1 = {5};  // OK: no narrowing needed
unsigned char uc2 = {-1}; // error: narrows
unsigned int ui1 = {-1};  // error: narrows
signed int si1 =
  { (unsigned int)-1 };   // error: narrows
int ii = {2.0};           // error: narrows
```

```
float f1 { x };          // error: might narrow
float f2 { 7 };          // OK: 7 can be exactly represented as a float
int f(int);
int a[] =
  { 2, f(2), f(2.0) };   // OK: the double-to-int conversion is not at the top level
```

— end example]


```
float f1 { x };          // error: might narrow
float f2 { 7 };          // OK: 7 can be exactly represented as a float
int f(int);
int a[] =
  { 2, f(2), f(2.0) };   // OK: the double-to-int conversion is not at the top level
```

— end example]

```
template <class T> struct Z {
  typedef typename T::x xx;
};
template <class T> typename Z<T>::xx f(void *, T); // #1
template <class T> void f(int, T);                 // #2
struct A {} a;
int main() {
  f(1, a);        // OK, deduction fails for #1 because there is no conversion from int to void*
}
```

— *end example*]

14.8.2.2 Deducing template arguments taking the address of a function template [temp.deduct.funcaddr]

1 Template arguments can be deduced from the type specified when taking the address of an overloaded function (13.4). The function template's function type and the specified type are used as the types of `P` and `A`, and the deduction is done as described in 14.8.2.5.

2 A placeholder type (7.1.7.4) in the return type of a function template is a non-deduced context. If template argument deduction succeeds for such a function, the return type is determined from instantiation of the function body.

14.8.2.3 Deducing conversion function template arguments [temp.deduct.conv]

1 Template argument deduction is done by comparing the return type of the conversion function template (call it `P`) with the type that is required as the result of the conversion (call it `A`; see 8.6, 13.3.1.5, and 13.3.1.6 for the determination of that type) as described in 14.8.2.5.

2 If `P` is a reference type, the type referred to by `P` is used in place of `P` for type deduction and for any further references to or transformations of `P` in the remainder of this section.

3 If `A` is not a reference type:

(3.1) — If `P` is an array type, the pointer type produced by the array-to-pointer standard conversion (4.2) is used in place of `P` for type deduction; otherwise,

(3.2) — If `P` is a function type, the pointer type produced by the function-to-pointer standard conversion (4.3) is used in place of `P` for type deduction; otherwise,

(3.3) — If `P` is a cv-qualified type, the top-level cv-qualifiers of `P`'s type are ignored for type deduction.

4 If `A` is a cv-qualified type, the top-level cv-qualifiers of `A`'s type are ignored for type deduction. If `A` is a reference type, the type referred to by `A` is used for type deduction.

5 In general, the deduction process attempts to find template argument values that will make the deduced `A` identical to `A`. However, there are four cases that allow a difference:

(5.1) — If the original `A` is a reference type, `A` can be more cv-qualified than the deduced `A` (i.e., the type referred to by the reference)

(5.2) — If the original `A` is a function pointer type, `A` can be "pointer to function" even if the deduced `A` is "pointer to noexcept function".

(5.3) — If the original `A` is a pointer to member function type, `A` can be "pointer to member of type function" even if the deduced `A` is "pointer to member of type noexcept function".

(5.4) — The deduced `A` can be another pointer or pointer to member type that can be converted to `A` via a qualification conversion.

6 These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced `A`, the type deduction fails.

©ISO/IEC Dxxxx

template <class T> struct Z {
  typedef typename T::x xx;
};
template <class T> typename Z<T>::xx f(void *, T); // #1
template <class T> void f(int, T);   // #2
struct A {} a;
int main() {
  f(1, a);   // OK, deduction fails for #1 because there is no conversion from int to void*
}

— end example]

14.8.2.2 Deducing template arguments taking the address of a function template [temp.deduct.funcaddr]

1 Template arguments can be deduced from the type specified when taking the address of an overloaded function (13.4). The function template's function type and the specified type are used as the types of P and A, and the deduction is done as described in 14.8.2.5.

2 A placeholder type (7.1.7.4) in the return type of a function template is a non-deduced context. If template argument deduction succeeds for such a function, the return type is determined from instantiation of the function body.

14.8.2.3 Deducing conversion function template arguments [temp.deduct.conv]

1 Template argument deduction is done by comparing the return type of the conversion function template (call it P) with the type that is required as the result of the conversion (call it A; see 8.6, 13.3.1.5, and 13.3.1.6 for the determination of that type) as described in 14.8.2.5.

2 If P is a reference type, the type referred to by P is used in place of P for type deduction and for any further references to or transformations of P in the remainder of this section.

3 If A is not a reference type:

(3.1) — If P is an array type, the pointer type produced by the array-to-pointer standard conversion (4.2) is used in place of P for type deduction; otherwise,

(3.2) — If P is a function type, the pointer type produced by the function-to-pointer standard conversion (4.3) is used in place of P for type deduction; otherwise,

(3.3) — If P is a cv-qualified type, the top-level cv-qualifiers of P's type are ignored for type deduction.

4 If A is a cv-qualified type, the top-level cv-qualifiers of A's type are ignored for type deduction. If A is a reference type, the type referred to by A is used for type deduction.

5 In general, the deduction process attempts to find template argument values that will make the deduced A identical to A. However, there are four cases that allow a difference:

(5.1) — If the original A is a reference type, A can be more cv-qualified than the deduced A (i.e., the type referred to by the reference)

(5.2) — If the original A is a function pointer type, A can be "pointer to function" even if the deduced A is "pointer to noexcept function".

(5.3) — If the original A is a pointer to member function type, A can be "pointer to member of type function" even if the deduced A is "pointer to member of type noexcept function".

(5.4) — The deduced A can be another pointer or pointer to member type that can be converted to A via a qualification conversion.

6 These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced A, the type deduction fails.

§ 14.8.2.3 407

18.2.4 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 International Standard. Use of the `offsetof` macro with a *type* other than a standard-layout class (Clause 9) is conditionally-supported.[187] The expression `offsetof(type, member-designator)` is never type-dependent (14.6.2.2) and it is value-dependent (14.6.2.3) 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 5.7.

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 (4.15) 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 POD 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: Alignment (3.11), Sizeof (5.3.3), Additive operators (5.7), Free store (12.5), and ISO C 7.19.

18.3 Implementation properties [support.limits]

18.3.1 General [support.limits.general]

1 The headers `<limits>` (18.3.2), `<climits>` (18.3.5), and `<cfloat>` (18.3.6) supply characteristics of implementation-dependent arithmetic types (3.9.1).

18.3.2 Header `<limits>` synopsis [limits.syn]

```
namespace std {
  // 18.3.3, floating-point type properties
  enum float_round_style;
  enum float_denorm_style;

  // 18.3.4, 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<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>;
```

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

©ISO/IEC Dxxxx

18.2.4 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 International Standard. Use of the offsetof macro with a type other than a standard-layout class (Clause 9) is conditionally-supported.[187] The expression offsetof(type, member-designator) is never type-dependent (14.6.2.2) and it is value-dependent (14.6.2.3) 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 5.7.

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 (4.15) 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 POD 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: Alignment (3.11), Sizeof (5.3.3), Additive operators (5.7), Free store (12.5), and ISO C 7.19.

18.3 Implementation properties [support.limits]

18.3.1 General [support.limits.general]

1 The headers <limits> (18.3.2), <climits> (18.3.5), and <cfloat> (18.3.6) supply characteristics of implementation-dependent arithmetic types (3.9.1).

18.3.2 Header <limits> synopsis [limits.syn]

namespace std {
  // 18.3.3, floating-point type properties
  enum float_round_style;
  enum float_denorm_style;

  // 18.3.4, 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<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>;

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

§ 18.3.2 476

```
// 20.2.3, swap
template <class T>
  void swap(T& a, T& b) noexcept(see below);
template <class T, size_t N>
  void swap(T (&a)[N], T (&b)[N]) noexcept(is_nothrow_swappable_v<T>);

// 20.2.4, exchange
template <class T, class U = T>
  T exchange(T& obj, U&& new_val);

// 20.2.5, forward/move
template <class T>
  constexpr T&& forward(remove_reference_t<T>& t) noexcept;
template <class T>
  constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
template <class T>
  constexpr remove_reference_t<T>&& move(T&&) noexcept;
template <class T>
  constexpr conditional_t<
      !is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&, T&&>
    move_if_noexcept(T& x) noexcept;

// 20.2.6, as_const
template <class T>
  constexpr add_const_t<T>& as_const(T& t) noexcept;
template <class T>
  void as_const(const T&&) = delete;

// 20.2.7, declval
template <class T>
  add_rvalue_reference_t<T> declval() noexcept;  // as unevaluated operand

// 20.3, Compile-time integer sequences
template<class T, T...>
  struct integer_sequence;
template<size_t... I>
  using index_sequence = integer_sequence<size_t, I...>;

template<class T, T N>
  using make_integer_sequence = integer_sequence<T, see below>;
template<size_t N>
  using make_index_sequence = make_integer_sequence<size_t, N>;

template<class... T>
  using index_sequence_for = make_index_sequence<sizeof...(T)>;

// 20.4, class template pair
template <class T1, class T2>
  struct pair;

// 20.4.3, pair specialized algorithms
template <class T1, class T2>
  constexpr bool operator==(const pair<T1, T2>&, const pair<T1, T2>&);
template <class T1, class T2>
  constexpr bool operator< (const pair<T1, T2>&, const pair<T1, T2>&);
```



```
// 20.2.3, swap
template <class T>
  void swap(T& a, T& b) noexcept(see below);
template <class T, size_t N>
  void swap(T (&a)[N], T (&b)[N]) noexcept(is_nothrow_swappable_v<T>);

// 20.2.4, exchange
template <class T, class U = T>
  T exchange(T& obj, U&& new_val);

// 20.2.5, forward/move
template <class T>
  constexpr T&& forward(remove_reference_t<T>& t) noexcept;
template <class T>
  constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
template <class T>
  constexpr remove_reference_t<T>&& move(T&&) noexcept;
template <class T>
  constexpr conditional_t<
      !is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&, T&&>
    move_if_noexcept(T& x) noexcept;

// 20.2.6, as_const
template <class T>
  constexpr add_const_t<T>& as_const(T& t) noexcept;
template <class T>
  void as_const(const T&&) = delete;

// 20.2.7, declval
template <class T>
  add_rvalue_reference_t<T> declval() noexcept;  // as unevaluated operand

// 20.3, Compile-time integer sequences
template<class T, T...>
  struct integer_sequence;
template<size_t... I>
  using index_sequence = integer_sequence<size_t, I...>;

template<class T, T N>
  using make_integer_sequence = integer_sequence<T, see below>;
template<size_t N>
  using make_index_sequence = make_integer_sequence<size_t, N>;

template<class... T>
  using index_sequence_for = make_index_sequence<sizeof...(T)>;

// 20.4, class template pair
template <class T1, class T2>
  struct pair;

// 20.4.3, pair specialized algorithms
template <class T1, class T2>
  constexpr bool operator==(const pair<T1, T2>&, const pair<T1, T2>&);
template <class T1, class T2>
  constexpr bool operator< (const pair<T1, T2>&, const pair<T1, T2>&);
```

```
template <class T, class... Types>
  constexpr const T&& get(const tuple<Types...>&& t) noexcept;
```

5 *Requires:* The type `T` occurs exactly once in `Types...`. Otherwise, the program is ill-formed.

6 *Returns:* A reference to the element of `t` corresponding to the type `T` in `Types...`.

7 [*Example:*

```
const tuple<int, const int, double, double> t(1, 2, 3.4, 5.6);
const int& i1 = get<int>(t);         // OK. Not ambiguous. i1 == 1
const int& i2 = get<const int>(t);   // OK. Not ambiguous. i2 == 2
const double& d = get<double>(t);    // ERROR. ill-formed
```

— *end example*]

8 [*Note:* The reason `get` is a non-member function is that if this functionality had been provided as a member function, code where the type depended on a template parameter would have required using the `template` keyword. — *end note*]

20.5.3.8 Relational operators [tuple.rel]

```
template<class... TTypes, class... UTypes>
  constexpr bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

1 *Requires:* For all `i`, where `0 <= i and i < sizeof...(TTypes)`, `get<i>(t) == get<i>(u)` is a valid expression returning a type that is convertible to `bool`. `sizeof...(TTypes) == sizeof...(UTypes)`.

2 *Returns:* `true` if `get<i>(t) == get<i>(u)` for all `i`, otherwise `false`. For any two zero-length tuples `e` and `f`, `e == f` returns `true`.

3 *Effects:* The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to `false`.

```
template<class... TTypes, class... UTypes>
  constexpr bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

4 *Requires:* For all `i`, where `0 <= i and i < sizeof...(TTypes)`, both `get<i>(t) < get<i>(u)` and `get<i>(u) < get<i>(t)` are valid expressions returning types that are convertible to `bool`. `sizeof...(TTypes) == sizeof...(UTypes)`.

5 *Returns:* The result of a lexicographical comparison between `t` and `u`. The result is defined as: `(bool)(get<0>(t) < get<0>(u)) || (!(bool)(get<0>(u) < get<0>(t)) && t`$_{\text{tail}}$ `<` u_{tail}`)`, where $\texttt{r}_{\text{tail}}$ for some tuple `r` is a tuple containing all but the first element of `r`. For any two zero-length tuples `e` and `f`, `e < f` returns `false`.

```
template<class... TTypes, class... UTypes>
  constexpr bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

6 *Returns:* `!(t == u)`.

```
template<class... TTypes, class... UTypes>
  constexpr bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

7 *Returns:* `u < t`.

```
template<class... TTypes, class... UTypes>
  constexpr bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

8 *Returns:* `!(u < t)`.

```
template<class... TTypes, class... UTypes>
```


©ISO/IEC Dxxxx

template <class T, class... Types>
constexpr const T&& get(const tuple<Types...>&& t) noexcept;

5 *Requires:* The type T occurs exactly once in Types.... Otherwise, the program is ill-formed.

6 *Returns:* A reference to the element of t corresponding to the type T in Types....

7 [*Example:*

const tuple<int, const int, double, double> t(1, 2, 3.4, 5.6);
const int& i1 = get<int>(t); // OK. Not ambiguous. i1 == 1
const int& i2 = get<const int>(t); // OK. Not ambiguous. i2 == 2
const double& d = get<double>(t); // ERROR. ill-formed

— *end example*]

8 [*Note:* The reason get is a non-member function is that if this functionality had been provided as a member function, code where the type depended on a template parameter would have required using the template keyword. — *end note*]

20.5.3.8 Relational operators [tuple.rel]

template<class... TTypes, class... UTypes>
constexpr bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

1 *Requires:* For all i, where 0 <= i and i < sizeof...(TTypes), get<i>(t) == get<i>(u) is a valid expression returning a type that is convertible to bool. sizeof...(TTypes) == sizeof...(UTypes).

2 *Returns:* true if get<i>(t) == get<i>(u) for all i, otherwise false. For any two zero-length tuples e and f, e == f returns true.

3 *Effects:* The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to false.

template<class... TTypes, class... UTypes>
constexpr bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

4 *Requires:* For all i, where 0 <= i and i < sizeof...(TTypes), both get<i>(t) < get<i>(u) and get<i>(u) < get<i>(t) are valid expressions returning types that are convertible to bool. sizeof...(TTypes) == sizeof...(UTypes).

5 *Returns:* The result of a lexicographical comparison between t and u. The result is defined as: (bool)(get<0>(t) < get<0>(u)) || (!(bool)(get<0>(u) < get<0>(t)) && t$_{\text{tail}}$ < u$_{\text{tail}}$), where r$_{\text{tail}}$ for some tuple r is a tuple containing all but the first element of r. For any two zero-length tuples e and f, e < f returns false.

template<class... TTypes, class... UTypes>
constexpr bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

6 *Returns:* !(t == u).

template<class... TTypes, class... UTypes>
constexpr bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

7 *Returns:* u < t.

template<class... TTypes, class... UTypes>
constexpr bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

8 *Returns:* !(u < t).

template<class... TTypes, class... UTypes>

§ 20.5.3.8 546

```
  void uninitialized_fill(ExecutionPolicy&& exec, // see 25.4.5
                          ForwardIterator first, ForwardIterator last,
                          const T& x);
template <class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
template <class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ExecutionPolicy&& exec, // see 25.4.5
                                       ForwardIterator first, Size n, const T& x);
template <class T>
  void destroy_at(T* location);
template <class ForwardIterator>
  void destroy(ForwardIterator first, ForwardIterator last);
template <class ExecutionPolicy, class ForwardIterator>
  void destroy(ExecutionPolicy&& exec, // see 25.4.5
               ForwardIterator first, ForwardIterator last);
template <class ForwardIterator, class Size>
  ForwardIterator destroy_n(ForwardIterator first, Size n);
template <class ExecutionPolicy, class ForwardIterator, class Size>
  ForwardIterator destroy_n(ExecutionPolicy&& exec, // see 25.4.5
                            ForwardIterator first, Size n);

// 20.11.1, class template unique_ptr
template <class T> struct default_delete;
template <class T> struct default_delete<T[]>;
template <class T, class D = default_delete<T>> class unique_ptr;
template <class T, class D> class unique_ptr<T[], D>;

template <class T, class... Args> unique_ptr<T> make_unique(Args&&... args);
template <class T> unique_ptr<T> make_unique(size_t n);
template <class T, class... Args> unspecified make_unique(Args&&...) = delete;

template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;

template <class T1, class D1, class T2, class D2>
  bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template <class T, class D>
  bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator==(nullptr_t, const unique_ptr<T, D>& y) noexcept;
template <class T, class D>
  bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator!=(nullptr_t, const unique_ptr<T, D>& y) noexcept;
template <class T, class D>
```



```
  void uninitialized_fill(ExecutionPolicy&& exec, // see 25.4.5
                          ForwardIterator first, ForwardIterator last,
                          const T& x);
template <class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
template <class ExecutionPolicy, class ForwardIterator, class Size, class T>
  ForwardIterator uninitialized_fill_n(ExecutionPolicy&& exec, // see 25.4.5
                                       ForwardIterator first, Size n, const T& x);
template <class T>
  void destroy_at(T* location);
template <class ForwardIterator>
  void destroy(ForwardIterator first, ForwardIterator last);
template <class ExecutionPolicy, class ForwardIterator>
  void destroy(ExecutionPolicy&& exec, // see 25.4.5
               ForwardIterator first, ForwardIterator last);
template <class ForwardIterator, class Size>
  ForwardIterator destroy_n(ForwardIterator first, Size n);
template <class ExecutionPolicy, class ForwardIterator, class Size>
  ForwardIterator destroy_n(ExecutionPolicy&& exec, // see 25.4.5
                            ForwardIterator first, Size n);

// 20.11.1, class template unique_ptr
template <class T> struct default_delete;
template <class T> struct default_delete<T[]>;
template <class T, class D = default_delete<T>> class unique_ptr;
template <class T, class D> class unique_ptr<T[], D>;

template <class T, class... Args> unique_ptr<T> make_unique(Args&&... args);
template <class T> unique_ptr<T> make_unique(size_t n);
template <class T, class... Args> unspecified make_unique(Args&&...) = delete;

template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;

template <class T1, class D1, class T2, class D2>
  bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
template <class T1, class D1, class T2, class D2>
  bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

template <class T, class D>
  bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator==(nullptr_t, const unique_ptr<T, D>& y) noexcept;
template <class T, class D>
  bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;
template <class T, class D>
  bool operator!=(nullptr_t, const unique_ptr<T, D>& y) noexcept;
template <class T, class D>
```

```
  bool operator<(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<=(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>=(nullptr_t, const unique_ptr<T, D>& y);

// 20.11.2.1, class bad_weak_ptr
class bad_weak_ptr;

// 20.11.2.2, class template shared_ptr
template<class T> class shared_ptr;

// 20.11.2.2.6, shared_ptr creation
template<class T, class... Args>
  shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// 20.11.2.2.7, shared_ptr comparisons
template<class T, class U>
  bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

template <class T>
  bool operator==(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator==(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator!=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator!=(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator<(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator<(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
```



```
  bool operator<(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator<=(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>(nullptr_t, const unique_ptr<T, D>& y);
template <class T, class D>
  bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
template <class T, class D>
  bool operator>=(nullptr_t, const unique_ptr<T, D>& y);

// 20.11.2.1, class bad_weak_ptr
class bad_weak_ptr;

// 20.11.2.2, class template shared_ptr
template<class T> class shared_ptr;

// 20.11.2.2.6, shared_ptr creation
template<class T, class... Args>
  shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// 20.11.2.2.7, shared_ptr comparisons
template<class T, class U>
  bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

template <class T>
  bool operator==(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator==(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator!=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator!=(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator<(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator<(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
```

```
  bool operator<=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator<=(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator>(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator>=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& y) noexcept;

// 20.11.2.2.8, shared_ptr specialized algorithms
template<class T>
  void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 20.11.2.2.9, shared_ptr casts
template<class T, class U>
  shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;

// 20.11.2.2.10, shared_ptr get_deleter
template<class D, class T>
  D* get_deleter(const shared_ptr<T>& p) noexcept;

// 20.11.2.2.11, shared_ptr I/O
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>& p);

// 20.11.2.3, class template weak_ptr
template<class T> class weak_ptr;

// 20.11.2.3.6, weak_ptr specialized algorithms
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

// 20.11.2.4, class template owner_less
template<class T = void> struct owner_less;

// 20.11.2.5, class template enable_shared_from_this
template<class T> class enable_shared_from_this;

// 20.11.2.6, shared_ptr atomic access
template<class T>
  bool atomic_is_lock_free(const shared_ptr<T>* p);

template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T>* p);
template<class T>
  shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);

template<class T>
  void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
```



```
  bool operator<=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator<=(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator>(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& y) noexcept;
template <class T>
  bool operator>=(const shared_ptr<T>& x, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& y) noexcept;

// 20.11.2.2.8, shared_ptr specialized algorithms
template<class T>
  void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 20.11.2.2.9, shared_ptr casts
template<class T, class U>
  shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;

// 20.11.2.2.10, shared_ptr get_deleter
template<class D, class T>
  D* get_deleter(const shared_ptr<T>& p) noexcept;

// 20.11.2.2.11, shared_ptr I/O
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>& p);

// 20.11.2.3, class template weak_ptr
template<class T> class weak_ptr;

// 20.11.2.3.6, weak_ptr specialized algorithms
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

// 20.11.2.4, class template owner_less
template<class T = void> struct owner_less;

// 20.11.2.5, class template enable_shared_from_this
template<class T> class enable_shared_from_this;

// 20.11.2.6, shared_ptr atomic access
template<class T>
  bool atomic_is_lock_free(const shared_ptr<T>* p);

template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T>* p);
template<class T>
  shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);

template<class T>
  void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
```

```
  template<class T>
    void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

  template<class T>
    shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
  template<class T>
    shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

  template<class T>
    bool atomic_compare_exchange_weak(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
    bool atomic_compare_exchange_strong(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
    bool atomic_compare_exchange_weak_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
      memory_order success, memory_order failure);
  template<class T>
    bool atomic_compare_exchange_strong_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
      memory_order success, memory_order failure);

  // 20.11.2.7, hash support
  template <class T> struct hash;
  template <class T, class D> struct hash<unique_ptr<T, D>>;
  template <class T> struct hash<shared_ptr<T>>;

  // 20.10.7.1, uses_allocator
  template <class T, class Alloc>
    constexpr bool uses_allocator_v = uses_allocator<T, Alloc>::value;
}
```

20.10.3 Pointer traits [pointer.traits]

1 The class template `pointer_traits` supplies a uniform interface to certain attributes of pointer-like types.

```
namespace std {
  template <class Ptr> struct pointer_traits {
    using pointer         = Ptr;
    using element_type    = see below;
    using difference_type = see below;

    template <class U> using rebind = see below;

    static pointer pointer_to(see below r);
  };

  template <class T> struct pointer_traits<T*> {
    using pointer         = T*;
    using element_type    = T;
    using difference_type = ptrdiff_t;

    template <class U> using rebind = U*;

    static pointer pointer_to(see below r) noexcept;
```



```
  template<class T>
    void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

  template<class T>
    shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
  template<class T>
    shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

  template<class T>
    bool atomic_compare_exchange_weak(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
    bool atomic_compare_exchange_strong(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
    bool atomic_compare_exchange_weak_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
      memory_order success, memory_order failure);
  template<class T>
    bool atomic_compare_exchange_strong_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
      memory_order success, memory_order failure);

  // 20.11.2.7, hash support
  template <class T> struct hash;
  template <class T, class D> struct hash<unique_ptr<T, D>>;
  template <class T> struct hash<shared_ptr<T>>;

  // 20.10.7.1, uses_allocator
  template <class T, class Alloc>
    constexpr bool uses_allocator_v = uses_allocator<T, Alloc>::value;
}
```

20.10.3 Pointer traits [pointer.traits]

1 The class template `pointer_traits` supplies a uniform interface to certain attributes of pointer-like types.

```
namespace std {
  template <class Ptr> struct pointer_traits {
    using pointer         = Ptr;
    using element_type    = see below;
    using difference_type = see below;

    template <class U> using rebind = see below;

    static pointer pointer_to(see below r);
  };

  template <class T> struct pointer_traits<T*> {
    using pointer         = T*;
    using element_type    = T;
    using difference_type = ptrdiff_t;

    template <class U> using rebind = U*;

    static pointer pointer_to(see below r) noexcept;
```

2 *Remarks:* This constructor shall not participate in overload resolution unless `U*` is implicitly convertible to `T*`.

```
void operator()(T* ptr) const;
```

3 *Effects:* Calls `delete` on `ptr`.

4 *Remarks:* If `T` is an incomplete type, the program is ill-formed.

20.11.1.1.3 `default_delete<T[]>` [unique.ptr.dltr.dflt1]

```
namespace std {
  template <class T> struct default_delete<T[]> {
    constexpr default_delete() noexcept = default;
    template <class U> default_delete(const default_delete<U[]>&) noexcept;
    template <class U> void operator()(U* ptr) const;
  };
}

template <class U> default_delete(const default_delete<U[]>& other) noexcept;
```

1 *Effects:* constructs a `default_delete` object from another `default_delete<U[]>` object.

2 *Remarks:* This constructor shall not participate in overload resolution unless `U(*)[]` is convertible to `T(*)[]`.

```
template <class U> void operator()(U* ptr) const;
```

3 *Effects:* Calls `delete[]` on `ptr`.

4 *Remarks:* If `U` is an incomplete type, the program is ill-formed. This function shall not participate in overload resolution unless `U(*)[]` is convertible to `T(*)[]`.

20.11.1.2 `unique_ptr` for single objects [unique.ptr.single]

```
namespace std {
  template <class T, class D = default_delete<T>> class unique_ptr {
  public:
    using pointer      = see below;
    using element_type = T;
    using deleter_type = D;

    // 20.11.1.2.1, constructors
    constexpr unique_ptr() noexcept;
    explicit unique_ptr(pointer p) noexcept;
    unique_ptr(pointer p, see below d1) noexcept;
    unique_ptr(pointer p, see below d2) noexcept;
    unique_ptr(unique_ptr&& u) noexcept;
    constexpr unique_ptr(nullptr_t) noexcept
      : unique_ptr() { }
    template <class U, class E>
      unique_ptr(unique_ptr<U, E>&& u) noexcept;

    // 20.11.1.2.2, destructor
    ~unique_ptr();

    // 20.11.1.2.3, assignment
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
    unique_ptr& operator=(nullptr_t) noexcept;
```


©ISO/IEC Dxxxx

2 *Remarks:* This constructor shall not participate in overload resolution unless `U*` is implicitly convertible to `T*`.

```
void operator()(T* ptr) const;
```

3 *Effects:* Calls `delete` on `ptr`.

4 *Remarks:* If `T` is an incomplete type, the program is ill-formed.

20.11.1.1.3 `default_delete<T[]>` [unique.ptr.dltr.dflt1]

```
namespace std {
  template <class T> struct default_delete<T[]> {
    constexpr default_delete() noexcept = default;
    template <class U> default_delete(const default_delete<U[]>&) noexcept;
    template <class U> void operator()(U* ptr) const;
  };
}

template <class U> default_delete(const default_delete<U[]>& other) noexcept;
```

1 *Effects:* constructs a `default_delete` object from another `default_delete<U[]>` object.

2 *Remarks:* This constructor shall not participate in overload resolution unless `U(*)[]` is convertible to `T(*)[]`.

```
template <class U> void operator()(U* ptr) const;
```

3 *Effects:* Calls `delete[]` on `ptr`.

4 *Remarks:* If `U` is an incomplete type, the program is ill-formed. This function shall not participate in overload resolution unless `U(*)[]` is convertible to `T(*)[]`.

20.11.1.2 `unique_ptr` for single objects [unique.ptr.single]

```
namespace std {
  template <class T, class D = default_delete<T>> class unique_ptr {
  public:
    using pointer      = see below;
    using element_type = T;
    using deleter_type = D;

    // 20.11.1.2.1, constructors
    constexpr unique_ptr() noexcept;
    explicit unique_ptr(pointer p) noexcept;
    unique_ptr(pointer p, see below d1) noexcept;
    unique_ptr(pointer p, see below d2) noexcept;
    unique_ptr(unique_ptr&& u) noexcept;
    constexpr unique_ptr(nullptr_t) noexcept
      : unique_ptr() { }
    template <class U, class E>
      unique_ptr(unique_ptr<U, E>&& u) noexcept;

    // 20.11.1.2.2, destructor
    ~unique_ptr();

    // 20.11.1.2.3, assignment
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
    unique_ptr& operator=(nullptr_t) noexcept;
```

```
  // 20.11.2.2.3, assignment
  shared_ptr& operator=(const shared_ptr& r) noexcept;
  template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;
  shared_ptr& operator=(shared_ptr&& r) noexcept;
  template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
  template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);

  // 20.11.2.2.4, modifiers
  void swap(shared_ptr& r) noexcept;
  void reset() noexcept;
  template<class Y> void reset(Y* p);
  template<class Y, class D> void reset(Y* p, D d);
  template<class Y, class D, class A> void reset(Y* p, D d, A a);

  // 20.11.2.2.5, observers
  element_type* get() const noexcept;
  T& operator*() const noexcept;
  T* operator->() const noexcept;
  element_type& operator[](ptrdiff_t i) const;
  long use_count() const noexcept;
  explicit operator bool() const noexcept;
  template<class U> bool owner_before(const shared_ptr<U>& b) const;
  template<class U> bool owner_before(const weak_ptr<U>& b) const;
};

// 20.11.2.2.6, shared_ptr creation
template<class T, class... Args>
  shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// 20.11.2.2.7, shared_ptr comparisons
template<class T, class U>
  bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

template <class T>
  bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator==(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator!=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
```



```
  // 20.11.2.2.3, assignment
  shared_ptr& operator=(const shared_ptr& r) noexcept;
  template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;
  shared_ptr& operator=(shared_ptr&& r) noexcept;
  template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
  template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);

  // 20.11.2.2.4, modifiers
  void swap(shared_ptr& r) noexcept;
  void reset() noexcept;
  template<class Y> void reset(Y* p);
  template<class Y, class D> void reset(Y* p, D d);
  template<class Y, class D, class A> void reset(Y* p, D d, A a);

  // 20.11.2.2.5, observers
  element_type* get() const noexcept;
  T& operator*() const noexcept;
  T* operator->() const noexcept;
  element_type& operator[](ptrdiff_t i) const;
  long use_count() const noexcept;
  explicit operator bool() const noexcept;
  template<class U> bool owner_before(const shared_ptr<U>& b) const;
  template<class U> bool owner_before(const weak_ptr<U>& b) const;
};

// 20.11.2.2.6, shared_ptr creation
template<class T, class... Args>
  shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// 20.11.2.2.7, shared_ptr comparisons
template<class T, class U>
  bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

template <class T>
  bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator==(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator!=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
```

```
  bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator<(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator<=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator<=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator>(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& b) noexcept;

// 20.11.2.2.8, shared_ptr specialized algorithms
template<class T>
  void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 20.11.2.2.9, shared_ptr casts
template<class T, class U>
  shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;

// 20.11.2.2.10, shared_ptr get_deleter
template<class D, class T>
  D* get_deleter(const shared_ptr<T>& p) noexcept;

// 20.11.2.2.11, shared_ptr I/O
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>& p);
}
```

2 Specializations of `shared_ptr` shall be `CopyConstructible`, `CopyAssignable`, and `LessThanComparable`, allowing their use in standard containers. Specializations of `shared_ptr` shall be contextually convertible to `bool`, allowing their use in boolean expressions and declarations in conditions. The template parameter `T` of `shared_ptr` may be an incomplete type.

3 [*Example:*

```
if (shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
  // do something with px
}
```

— *end example*]

4 For purposes of determining the presence of a data race, member functions shall access and modify only the `shared_ptr` and `weak_ptr` objects themselves and not objects they refer to. Changes in `use_count()` do not reflect modifications that can introduce data races.

5 For the purposes of subclause 20.11.2, a pointer type `Y*` is said to be *compatible with* a pointer type `T*` when either `Y*` is convertible to `T*` or `Y` is `U[N]` and `T` is *cv* `U[]`.


```
  bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator<(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator<=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator<=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator>(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& b) noexcept;

// 20.11.2.2.8, shared_ptr specialized algorithms
template<class T>
  void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 20.11.2.2.9, shared_ptr casts
template<class T, class U>
  shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;

// 20.11.2.2.10, shared_ptr get_deleter
template<class D, class T>
  D* get_deleter(const shared_ptr<T>& p) noexcept;

// 20.11.2.2.11, shared_ptr I/O
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>& p);
}
```

2 Specializations of `shared_ptr` shall be `CopyConstructible`, `CopyAssignable`, and `LessThanComparable`, allowing their use in standard containers. Specializations of `shared_ptr` shall be contextually convertible to `bool`, allowing their use in boolean expressions and declarations in conditions. The template parameter `T` of `shared_ptr` may be an incomplete type.

3 [*Example:*

```
if (shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
  // do something with px
}
```

— *end example*]

4 For purposes of determining the presence of a data race, member functions shall access and modify only the `shared_ptr` and `weak_ptr` objects themselves and not objects they refer to. Changes in `use_count()` do not reflect modifications that can introduce data races.

5 For the purposes of subclause 20.11.2, a pointer type `Y*` is said to be *compatible with* a pointer type `T*` when either `Y*` is convertible to `T*` or `Y` is `U[N]` and `T` is *cv* `U[]`.

20.13.5 Scoped allocator operators [scoped.adaptor.operators]

```
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
```

1 *Returns:* If `sizeof...(InnerAllocs)` is zero,

```
a.outer_allocator() == b.outer_allocator()
```

otherwise

```
a.outer_allocator() == b.outer_allocator() && a.inner_allocator() == b.inner_allocator()
```

```
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;
```

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

20.14 Function objects [function.objects]

1 A *function object type* is an object type (3.9) that can be the type of the *postfix-expression* in a function call (5.2.2, 13.3.1.1).[222] A *function object* is an object of a function object type. In the places where one would expect to pass a pointer to a function to an algorithmic template (Clause 25), the interface is specified to accept a function object. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

20.14.1 Header <functional> synopsis [functional.syn]

```
namespace std {
  // 20.14.4, invoke
  template <class F, class... Args>
    result_of_t<F&&(Args&&...)> invoke(F&& f, Args&&... args);

  // 20.14.5, reference_wrapper
  template <class T> class reference_wrapper;

  template <class T> reference_wrapper<T> ref(T&) noexcept;
  template <class T> reference_wrapper<const T> cref(const T&) noexcept;
  template <class T> void ref(const T&&) = delete;
  template <class T> void cref(const T&&) = delete;

  template <class T> reference_wrapper<T> ref(reference_wrapper<T>) noexcept;
  template <class T> reference_wrapper<const T> cref(reference_wrapper<T>) noexcept;

  // 20.14.6, arithmetic operations
  template <class T = void> struct plus;
  template <class T = void> struct minus;
  template <class T = void> struct multiplies;
  template <class T = void> struct divides;
  template <class T = void> struct modulus;
  template <class T = void> struct negate;
  template <> struct plus<void>;
  template <> struct minus<void>;
  template <> struct multiplies<void>;
  template <> struct divides<void>;
```

222) Such a type is a function pointer or a class type which has a member `operator()` or a class type which has a conversion to a pointer to function.

©ISO/IEC Dxxxx

20.13.5 Scoped allocator operators [scoped.adaptor.operators]

template <class OuterA1, class OuterA2, class... InnerAllocs>
bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;

1 Returns: If sizeof...(InnerAllocs) is zero,

a.outer_allocator() == b.outer_allocator()

otherwise

a.outer_allocator() == b.outer_allocator() && a.inner_allocator() == b.inner_allocator()

template <class OuterA1, class OuterA2, class... InnerAllocs>
bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;

2 Returns: !(a == b).

20.14 Function objects [function.objects]

1 A function object type is an object type (3.9) that can be the type of the postfix-expression in a function call (5.2.2, 13.3.1.1).[222] A function object is an object of a function object type. In the places where one would expect to pass a pointer to a function to an algorithmic template (Clause 25), the interface is specified to accept a function object. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

20.14.1 Header <functional> synopsis [functional.syn]

namespace std {
// 20.14.4, invoke
template <class F, class... Args>
result_of_t<F&&(Args&&...)> invoke(F&& f, Args&&... args);

// 20.14.5, reference_wrapper
template <class T> class reference_wrapper;

template <class T> reference_wrapper<T> ref(T&) noexcept;
template <class T> reference_wrapper<const T> cref(const T&) noexcept;
template <class T> void ref(const T&&) = delete;
template <class T> void cref(const T&&) = delete;

template <class T> reference_wrapper<T> ref(reference_wrapper<T>) noexcept;
template <class T> reference_wrapper<const T> cref(reference_wrapper<T>) noexcept;

// 20.14.6, arithmetic operations
template <class T = void> struct plus;
template <class T = void> struct minus;
template <class T = void> struct multiplies;
template <class T = void> struct divides;
template <class T = void> struct modulus;
template <class T = void> struct negate;
template <> struct plus<void>;
template <> struct minus<void>;
template <> struct multiplies<void>;
template <> struct divides<void>;

222) Such a type is a function pointer or a class type which has a member operator() or a class type which has a conversion to a pointer to function.

§ 20.14.1 644

```
  : bool_constant<see below> { };
```

1 If R1::num × R2::den is less than R2::num × R1::den, ratio_less<R1, R2> shall be derived from bool_constant<true>; otherwise it shall be derived from bool_constant<false>. Implementations may use other algorithms to compute this relationship to avoid overflow. If overflow occurs, the program is ill-formed.

```
template <class R1, class R2> struct ratio_less_equal
  : bool_constant<!ratio_less_v<R2, R1>> { };

template <class R1, class R2> struct ratio_greater
  : bool_constant<ratio_less_v<R2, R1>> { };

template <class R1, class R2> struct ratio_greater_equal
  : bool_constant<!ratio_less_v<R1, R2>> { };
```

20.16.6 SI types for ratio [ratio.si]

1 For each of the *typedef-names* yocto, zepto, zetta, and yotta, if both of the constants used in its specification are representable by intmax_t, the typedef shall be defined; if either of the constants is not representable by intmax_t, the typedef shall not be defined.

20.17 Time utilities [time]

20.17.1 In general [time.general]

1 This subclause describes the chrono library (20.17.2) and various C functions (20.17.8) that provide generally useful time utilities.

20.17.2 Header <chrono> synopsis [time.syn]

```
namespace std {
  namespace chrono {
    // 20.17.5, class template duration
    template <class Rep, class Period = ratio<1>> class duration;

    // 20.17.6, class template time_point
    template <class Clock, class Duration = typename Clock::duration> class time_point;
  }

  // 20.17.4.3, common_type specializations
  template <class Rep1, class Period1, class Rep2, class Period2>
    struct common_type<chrono::duration<Rep1, Period1>,
                       chrono::duration<Rep2, Period2>>;

  template <class Clock, class Duration1, class Duration2>
    struct common_type<chrono::time_point<Clock, Duration1>,
                       chrono::time_point<Clock, Duration2>>;

  namespace chrono {
    // 20.17.4, customization traits
    template <class Rep> struct treat_as_floating_point;
    template <class Rep> struct duration_values;
    template <class Rep> constexpr bool treat_as_floating_point_v
      = treat_as_floating_point<Rep>::value;

    // 20.17.5.5, duration arithmetic
    template <class Rep1, class Period1, class Rep2, class Period2>
```


©ISO/IEC Dxxxx

```
  : bool_constant<see below> { };
```

1 If R1::num × R2::den is less than R2::num × R1::den, ratio_less<R1, R2> shall be derived from bool_constant<true>; otherwise it shall be derived from bool_constant<false>. Implementations may use other algorithms to compute this relationship to avoid overflow. If overflow occurs, the program is ill-formed.

```
template <class R1, class R2> struct ratio_less_equal
  : bool_constant<!ratio_less_v<R2, R1>> { };

template <class R1, class R2> struct ratio_greater
  : bool_constant<ratio_less_v<R2, R1>> { };

template <class R1, class R2> struct ratio_greater_equal
  : bool_constant<!ratio_less_v<R1, R2>> { };
```

20.16.6 SI types for ratio [ratio.si]

1 For each of the *typedef-names* yocto, zepto, zetta, and yotta, if both of the constants used in its specification are representable by intmax_t, the typedef shall be defined; if either of the constants is not representable by intmax_t, the typedef shall not be defined.

20.17 Time utilities [time]

20.17.1 In general [time.general]

1 This subclause describes the chrono library (20.17.2) and various C functions (20.17.8) that provide generally useful time utilities.

20.17.2 Header <chrono> synopsis [time.syn]

```
namespace std {
  namespace chrono {
    // 20.17.5, class template duration
    template <class Rep, class Period = ratio<1>> class duration;

    // 20.17.6, class template time_point
    template <class Clock, class Duration = typename Clock::duration> class time_point;
  }

  // 20.17.4.3, common_type specializations
  template <class Rep1, class Period1, class Rep2, class Period2>
    struct common_type<chrono::duration<Rep1, Period1>,
                       chrono::duration<Rep2, Period2>>;

  template <class Clock, class Duration1, class Duration2>
    struct common_type<chrono::time_point<Clock, Duration1>,
                       chrono::time_point<Clock, Duration2>>;

  namespace chrono {
    // 20.17.4, customization traits
    template <class Rep> struct treat_as_floating_point;
    template <class Rep> struct duration_values;
    template <class Rep> constexpr bool treat_as_floating_point_v
      = treat_as_floating_point<Rep>::value;

    // 20.17.5.5, duration arithmetic
    template <class Rep1, class Period1, class Rep2, class Period2>
```

```
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator+(const duration<Rep1, Period1>& lhs,
                      const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator-(const duration<Rep1, Period1>& lhs,
                      const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period, class Rep2>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator*(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Rep2, class Period>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator*(const Rep1& s, const duration<Rep2, Period>& d);
template <class Rep1, class Period, class Rep2>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator/(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<Rep1, Rep2>
  constexpr operator/(const duration<Rep1, Period1>& lhs,
                      const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period, class Rep2>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator%(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator%(const duration<Rep1, Period1>& lhs,
                      const duration<Rep2, Period2>& rhs);

// 20.17.5.6, duration comparisons
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator==(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator!=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator< (const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator<=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator> (const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator>=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);

// 20.17.5.7, duration_cast
template <class ToDuration, class Rep, class Period>
  constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
template <class ToDuration, class Rep, class Period>
  constexpr ToDuration floor(const duration<Rep, Period>& d);
template <class ToDuration, class Rep, class Period>
  constexpr ToDuration ceil(const duration<Rep, Period>& d);
```



```
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator+(const duration<Rep1, Period1>& lhs,
                      const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator-(const duration<Rep1, Period1>& lhs,
                      const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period, class Rep2>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator*(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Rep2, class Period>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator*(const Rep1& s, const duration<Rep2, Period>& d);
template <class Rep1, class Period, class Rep2>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator/(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<Rep1, Rep2>
  constexpr operator/(const duration<Rep1, Period1>& lhs,
                      const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period, class Rep2>
  duration<common_type_t<Rep1, Rep2>, Period>
  constexpr operator%(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Period1, class Rep2, class Period2>
  common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
  constexpr operator%(const duration<Rep1, Period1>& lhs,
                      const duration<Rep2, Period2>& rhs);

// 20.17.5.6, duration comparisons
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator==(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator!=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator< (const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator<=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator> (const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  constexpr bool operator>=(const duration<Rep1, Period1>& lhs,
                            const duration<Rep2, Period2>& rhs);

// 20.17.5.7, duration_cast
template <class ToDuration, class Rep, class Period>
  constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
template <class ToDuration, class Rep, class Period>
  constexpr ToDuration floor(const duration<Rep, Period>& d);
template <class ToDuration, class Rep, class Period>
  constexpr ToDuration ceil(const duration<Rep, Period>& d);
```

```
template <class ToDuration, class Rep, class Period>
  constexpr ToDuration round(const duration<Rep, Period>& d);

// convenience typedefs
using nanoseconds  = duration<signed integer type of at least 64 bits, nano>;
using microseconds = duration<signed integer type of at least 55 bits, micro>;
using milliseconds = duration<signed integer type of at least 45 bits, milli>;
using seconds      = duration<signed integer type of at least 35 bits>;
using minutes      = duration<signed integer type of at least 29 bits, ratio<  60>>;
using hours        = duration<signed integer type of at least 23 bits, ratio<3600>>;

// 20.17.6.5, time_point arithmetic
template <class Clock, class Duration1, class Rep2, class Period2>
  constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
  operator+(const time_point<Clock, Duration1>& lhs,
            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Clock, class Duration2>
  constexpr time_point<Clock, common_type_t<duration<Rep1, Period1>, Duration2>>
  operator+(const duration<Rep1, Period1>& lhs,
            const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Rep2, class Period2>
  constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
  operator-(const time_point<Clock, Duration1>& lhs,
            const duration<Rep2, Period2>& rhs);
template <class Clock, class Duration1, class Duration2>
  constexpr common_type_t<Duration1, Duration2>
  operator-(const time_point<Clock, Duration1>& lhs,
            const time_point<Clock, Duration2>& rhs);

// 20.17.6.6, time_point comparisons
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator==(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator!=(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator< (const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator<=(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator> (const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator>=(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);

// 20.17.6.7, time_point_cast
template <class ToDuration, class Clock, class Duration>
  constexpr time_point<Clock, ToDuration>
  time_point_cast(const time_point<Clock, Duration>& t);
template <class ToDuration, class Clock, class Duration>
  constexpr time_point<Clock, ToDuration>
```



```
template <class ToDuration, class Rep, class Period>
  constexpr ToDuration round(const duration<Rep, Period>& d);

// convenience typedefs
using nanoseconds  = duration<signed integer type of at least 64 bits, nano>;
using microseconds = duration<signed integer type of at least 55 bits, micro>;
using milliseconds = duration<signed integer type of at least 45 bits, milli>;
using seconds      = duration<signed integer type of at least 35 bits>;
using minutes      = duration<signed integer type of at least 29 bits, ratio<  60>>;
using hours        = duration<signed integer type of at least 23 bits, ratio<3600>>;

// 20.17.6.5, time_point arithmetic
template <class Clock, class Duration1, class Rep2, class Period2>
  constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
  operator+(const time_point<Clock, Duration1>& lhs,
            const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Clock, class Duration2>
  constexpr time_point<Clock, common_type_t<duration<Rep1, Period1>, Duration2>>
  operator+(const duration<Rep1, Period1>& lhs,
            const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Rep2, class Period2>
  constexpr time_point<Clock, common_type_t<Duration1, duration<Rep2, Period2>>>
  operator-(const time_point<Clock, Duration1>& lhs,
            const duration<Rep2, Period2>& rhs);
template <class Clock, class Duration1, class Duration2>
  constexpr common_type_t<Duration1, Duration2>
  operator-(const time_point<Clock, Duration1>& lhs,
            const time_point<Clock, Duration2>& rhs);

// 20.17.6.6, time_point comparisons
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator==(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator!=(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator< (const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator<=(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator> (const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   constexpr bool operator>=(const time_point<Clock, Duration1>& lhs,
                             const time_point<Clock, Duration2>& rhs);

// 20.17.6.7, time_point_cast
template <class ToDuration, class Clock, class Duration>
  constexpr time_point<Clock, ToDuration>
  time_point_cast(const time_point<Clock, Duration>& t);
template <class ToDuration, class Clock, class Duration>
  constexpr time_point<Clock, ToDuration>
```

6 *Returns:* A `duration` literal representing `minutes` minutes.

```
constexpr chrono::seconds               operator "" s(unsigned long long sec);
constexpr chrono::duration<unspecified> operator "" s(long double sec);
```

7 *Returns:* A `duration` literal representing `sec` seconds.

8 [*Note:* The same suffix `s` is used for `basic_string` but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. — *end note*]

```
constexpr chrono::milliseconds                operator "" ms(unsigned long long msec);
constexpr chrono::duration<unspecified, milli> operator "" ms(long double msec);
```

9 *Returns:* A `duration` literal representing `msec` milliseconds.

```
constexpr chrono::microseconds                operator "" us(unsigned long long usec);
constexpr chrono::duration<unspecified, micro> operator "" us(long double usec);
```

10 *Returns:* A `duration` literal representing `usec` microseconds.

```
constexpr chrono::nanoseconds                operator "" ns(unsigned long long nsec);
constexpr chrono::duration<unspecified, nano> operator "" ns(long double nsec);
```

11 *Returns:* A `duration` literal representing `nsec` nanoseconds.

20.17.5.9 `duration` algorithms [time.duration.alg]

```
template <class Rep, class Period>
  constexpr duration<Rep, Period> abs(duration<Rep, Period> d);
```

1 *Remarks:* This function shall not participate in overload resolution unless `numeric_limits<Rep>::is_signed` is `true`.

2 *Returns:* If `d >= d.zero()`, return `d`, otherwise return `-d`.

20.17.6 Class template `time_point` [time.point]

```
template <class Clock, class Duration = typename Clock::duration>
class time_point {
public:
  using clock    = Clock;
  using duration = Duration;
  using rep      = typename duration::rep;
  using period   = typename duration::period;
private:
  duration d_;  // exposition only

public:
  // 20.17.6.1, construct
  constexpr time_point();  // has value epoch
  constexpr explicit time_point(const duration& d);  // same as time_point() + d
  template <class Duration2>
    constexpr time_point(const time_point<clock, Duration2>& t);

  // 20.17.6.2, observer
  constexpr duration time_since_epoch() const;

  // 20.17.6.3, arithmetic
  constexpr time_point& operator+=(const duration& d);
  constexpr time_point& operator-=(const duration& d);
```


©ISO/IEC Dxxxx

6 *Returns:* A `duration` literal representing `minutes` minutes.

```
constexpr chrono::seconds               operator "" s(unsigned long long sec);
constexpr chrono::duration<unspecified> operator "" s(long double sec);
```

7 *Returns:* A `duration` literal representing `sec` seconds.

8 [*Note:* The same suffix `s` is used for `basic_string` but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. — *end note*]

```
constexpr chrono::milliseconds                operator "" ms(unsigned long long msec);
constexpr chrono::duration<unspecified, milli> operator "" ms(long double msec);
```

9 *Returns:* A `duration` literal representing `msec` milliseconds.

```
constexpr chrono::microseconds                operator "" us(unsigned long long usec);
constexpr chrono::duration<unspecified, micro> operator "" us(long double usec);
```

10 *Returns:* A `duration` literal representing `usec` microseconds.

```
constexpr chrono::nanoseconds                operator "" ns(unsigned long long nsec);
constexpr chrono::duration<unspecified, nano> operator "" ns(long double nsec);
```

11 *Returns:* A `duration` literal representing `nsec` nanoseconds.

20.17.5.9 `duration` algorithms [time.duration.alg]

```
template <class Rep, class Period>
  constexpr duration<Rep, Period> abs(duration<Rep, Period> d);
```

1 *Remarks:* This function shall not participate in overload resolution unless `numeric_limits<Rep>::is_signed` is `true`.

2 *Returns:* If `d >= d.zero()`, return `d`, otherwise return `-d`.

20.17.6 Class template `time_point` [time.point]

```
template <class Clock, class Duration = typename Clock::duration>
class time_point {
public:
  using clock    = Clock;
  using duration = Duration;
  using rep      = typename duration::rep;
  using period   = typename duration::period;
private:
  duration d_;  // exposition only

public:
  // 20.17.6.1, construct
  constexpr time_point();  // has value epoch
  constexpr explicit time_point(const duration& d);  // same as time_point() + d
  template <class Duration2>
    constexpr time_point(const time_point<clock, Duration2>& t);

  // 20.17.6.2, observer
  constexpr duration time_since_epoch() const;

  // 20.17.6.3, arithmetic
  constexpr time_point& operator+=(const duration& d);
  constexpr time_point& operator-=(const duration& d);
```

§ 20.17.6 708

5 The member `eof()` shall return `WEOF`.

21.3 String classes [string.classes]

1 The header `<string>` defines the `basic_string` class template for manipulating varying-length sequences of char-like objects and four *typedef-names*, `string`, `u16string`, `u32string`, and `wstring`, that name the specializations `basic_string<char>`, `basic_string<char16_t>`, `basic_string<char32_t>`, and `basic_string<wchar_t>`, respectively.

21.3.1 Header <string> synopsis [string.syn]

```
#include <initializer_list>

namespace std {
  // 21.2, character traits
  template<class charT> struct char_traits;
  template<> struct char_traits<char>;
  template<> struct char_traits<char16_t>;
  template<> struct char_traits<char32_t>;
  template<> struct char_traits<wchar_t>;

  // 21.3.2, basic_string
  template<class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT>>
      class basic_string;

  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(const basic_string<charT, traits, Allocator>& lhs,
                const basic_string<charT, traits, Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(basic_string<charT, traits, Allocator>&& lhs,
                const basic_string<charT, traits, Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(const basic_string<charT, traits, Allocator>& lhs,
                basic_string<charT, traits, Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(basic_string<charT, traits, Allocator>&& lhs,
                basic_string<charT, traits, Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(const charT* lhs,
                const basic_string<charT, traits, Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(const charT* lhs,
                basic_string<charT, traits, Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(charT lhs, const basic_string<charT, traits, Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(charT lhs, basic_string<charT, traits, Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
```


5 The member `eof()` shall return `WEOF`.

21.3 String classes [string.classes]

1 The header `<string>` defines the `basic_string` class template for manipulating varying-length sequences of char-like objects and four *typedef-names*, `string`, `u16string`, `u32string`, and `wstring`, that name the specializations `basic_string<char>`, `basic_string<char16_t>`, `basic_string<char32_t>`, and `basic_string<wchar_t>`, respectively.

21.3.1 Header <string> synopsis [string.syn]

```
#include <initializer_list>

namespace std {
  // 21.2, character traits
  template<class charT> struct char_traits;
  template<> struct char_traits<char>;
  template<> struct char_traits<char16_t>;
  template<> struct char_traits<char32_t>;
  template<> struct char_traits<wchar_t>;

  // 21.3.2, basic_string
  template<class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT>>
      class basic_string;

  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(const basic_string<charT, traits, Allocator>& lhs,
                const basic_string<charT, traits, Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(basic_string<charT, traits, Allocator>&& lhs,
                const basic_string<charT, traits, Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(const basic_string<charT, traits, Allocator>& lhs,
                basic_string<charT, traits, Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(basic_string<charT, traits, Allocator>&& lhs,
                basic_string<charT, traits, Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(const charT* lhs,
                const basic_string<charT, traits, Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(const charT* lhs,
                basic_string<charT, traits, Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(charT lhs, const basic_string<charT, traits, Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
      operator+(charT lhs, basic_string<charT, traits, Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
```

```
  inline namespace literals {
  inline namespace string_literals {
    // 21.3.6, suffix for basic_string literals
    string    operator "" s(const char* str, size_t len);
    u16string operator "" s(const char16_t* str, size_t len);
    u32string operator "" s(const char32_t* str, size_t len);
    wstring   operator "" s(const wchar_t* str, size_t len);
  }
  }
}
```

21.3.2 Class template basic_string [basic.string]

1 The class template basic_string describes objects that can store a sequence consisting of a varying number of arbitrary char-like objects with the first element of the sequence at position zero. Such a sequence is also called a "string" if the type of the char-like objects that it holds is clear from context. In the rest of this Clause, the type of the char-like objects held in a basic_string object is designated by charT.

2 The member functions of basic_string use an object of the Allocator class passed as a template parameter to allocate and free storage for the contained char-like objects.[225]

3 A basic_string is a contiguous container (23.2.1).

4 In all cases, size() <= capacity().

5 The functions described in this Clause can report two kinds of errors, each associated with an exception type:

(5.1) — a *length* error is associated with exceptions of type length_error (19.2.5);

(5.2) — an *out-of-range* error is associated with exceptions of type out_of_range (19.2.6).

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_string {
  public:
    // types:
    using traits_type            =          traits;
    using value_type             = typename traits::char_type;
    using allocator_type         = Allocator;
    using size_type              = typename allocator_traits<Allocator>::size_type;
    using difference_type        = typename allocator_traits<Allocator>::difference_type;
    using pointer                = typename allocator_traits<Allocator>::pointer;
    using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;

    using iterator               = implementation-defined; // see 23.2
    using const_iterator         = implementation-defined; // see 23.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    static const size_type npos  = -1;

    // 21.3.2.2, construct/copy/destroy
    basic_string() noexcept(noexcept(Allocator())) : basic_string(Allocator()) { }
    explicit basic_string(const Allocator& a) noexcept;
    basic_string(const basic_string& str);
```

225) Allocator::value_type must name the same type as charT (21.3.2.1).

©ISO/IEC Dxxxx


```
  inline namespace literals {
  inline namespace string_literals {
    // 21.3.6, suffix for basic_string literals
    string    operator "" s(const char* str, size_t len);
    u16string operator "" s(const char16_t* str, size_t len);
    u32string operator "" s(const char32_t* str, size_t len);
    wstring   operator "" s(const wchar_t* str, size_t len);
  }
  }
}
```

21.3.2 Class template basic_string [basic.string]

1 The class template basic_string describes objects that can store a sequence consisting of a varying number of arbitrary char-like objects with the first element of the sequence at position zero. Such a sequence is also called a "string" if the type of the char-like objects that it holds is clear from context. In the rest of this Clause, the type of the char-like objects held in a basic_string object is designated by charT.

2 The member functions of basic_string use an object of the Allocator class passed as a template parameter to allocate and free storage for the contained char-like objects.[225]

3 A basic_string is a contiguous container (23.2.1).

4 In all cases, size() <= capacity().

5 The functions described in this Clause can report two kinds of errors, each associated with an exception type:

(5.1) — a *length* error is associated with exceptions of type length_error (19.2.5);

(5.2) — an *out-of-range* error is associated with exceptions of type out_of_range (19.2.6).

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_string {
  public:
    // types:
    using traits_type            =          traits;
    using value_type             = typename traits::char_type;
    using allocator_type         = Allocator;
    using size_type              = typename allocator_traits<Allocator>::size_type;
    using difference_type        = typename allocator_traits<Allocator>::difference_type;
    using pointer                = typename allocator_traits<Allocator>::pointer;
    using const_pointer          = typename allocator_traits<Allocator>::const_pointer;
    using reference              = value_type&;
    using const_reference        = const value_type&;

    using iterator               = implementation-defined; // see 23.2
    using const_iterator         = implementation-defined; // see 23.2
    using reverse_iterator       = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    static const size_type npos  = -1;

    // 21.3.2.2, construct/copy/destroy
    basic_string() noexcept(noexcept(Allocator())) : basic_string(Allocator()) { }
    explicit basic_string(const Allocator& a) noexcept;
    basic_string(const basic_string& str);
```

225) Allocator::value_type must name the same type as charT (21.3.2.1).

a const rvalue of `X::value_type`, and `rv` denotes a non-const rvalue of `X::value_type`. `Args` denotes a template parameter pack; `args` denotes a function parameter pack with the pattern `Args&&`.

4 The complexities of the expressions are sequence dependent.

Table 87 — Sequence container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition
`X(n, t)` `X u(n, t);`		*Requires:* `T` shall be `CopyInsertable` into `X`. *Postconditions:* `distance(begin(), end()) == n` Constructs a sequence container with `n` copies of `t`
`X(i, j)` `X u(i, j);`		*Requires:* `T` shall be `EmplaceConstructible` into `X` from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (24.2.5), `T` shall also be `MoveInsertable` into `X`. Each iterator in the range `[i, j)` shall be dereferenced exactly once. *Postconditions:* `distance(begin(), end()) == distance(i, j)` Constructs a sequence container equal to the range `[i, j)`
`X(il)`		Equivalent to `X(il.begin(), il.end())`
`a = il`	`X&`	*Requires:* `T` is `CopyInsertable` into `X` and `CopyAssignable`. Assigns the range `[il.begin(), il.end())` into `a`. All existing elements of `a` are either assigned to or destroyed. *Returns:* `*this`.
`a.emplace(p, args)`	`iterator`	*Requires:* `T` is `EmplaceConstructible` into `X` from `args`. For `vector` and `deque`, `T` is also `MoveInsertable` into `X` and `MoveAssignable`. *Effects:* Inserts an object of type `T` constructed with `std::forward<Args>(args)...` before `p`.
`a.insert(p,t)`	`iterator`	*Requires:* `T` shall be `CopyInsertable` into `X`. For `vector` and `deque`, `T` shall also be `CopyAssignable`. *Effects:* Inserts a copy of `t` before `p`.
`a.insert(p,rv)`	`iterator`	*Requires:* `T` shall be `MoveInsertable` into `X`. For `vector` and `deque`, `T` shall also be `MoveAssignable`. *Effects:* Inserts a copy of `rv` before `p`.
`a.insert(p,n,t)`	`iterator`	*Requires:* `T` shall be `CopyInsertable` into `X` and `CopyAssignable`. Inserts `n` copies of `t` before `p`.

©ISO/IEC Dxxxx

a const rvalue of `X::value_type`, and `rv` denotes a non-const rvalue of `X::value_type`. `Args` denotes a template parameter pack; `args` denotes a function parameter pack with the pattern `Args&&`.

4 The complexities of the expressions are sequence dependent.

Table 87 — Sequence container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition
`X(n, t)` `X u(n, t);`		*Requires:* `T` shall be `CopyInsertable` into `X`. *Postconditions:* `distance(begin(), end()) == n` Constructs a sequence container with `n` copies of `t`
`X(i, j)` `X u(i, j);`		*Requires:* `T` shall be `EmplaceConstructible` into `X` from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (24.2.5), `T` shall also be `MoveInsertable` into `X`. Each iterator in the range `[i, j)` shall be dereferenced exactly once. *Postconditions:* `distance(begin(), end()) == distance(i, j)` Constructs a sequence container equal to the range `[i, j)`
`X(il)`		Equivalent to `X(il.begin(), il.end())`
`a = il`	`X&`	*Requires:* `T` is `CopyInsertable` into `X` and `CopyAssignable`. Assigns the range `[il.begin(), il.end())` into `a`. All existing elements of `a` are either assigned to or destroyed. *Returns:* `*this`.
`a.emplace(p, args)`	`iterator`	*Requires:* `T` is `EmplaceConstructible` into `X` from `args`. For `vector` and `deque`, `T` is also `MoveInsertable` into `X` and `MoveAssignable`. *Effects:* Inserts an object of type `T` constructed with `std::forward<Args>(args)...` before `p`.
`a.insert(p,t)`	`iterator`	*Requires:* `T` shall be `CopyInsertable` into `X`. For `vector` and `deque`, `T` shall also be `CopyAssignable`. *Effects:* Inserts a copy of `t` before `p`.
`a.insert(p,rv)`	`iterator`	*Requires:* `T` shall be `MoveInsertable` into `X`. For `vector` and `deque`, `T` shall also be `MoveAssignable`. *Effects:* Inserts a copy of `rv` before `p`.
`a.insert(p,n,t)`	`iterator`	*Requires:* `T` shall be `CopyInsertable` into `X` and `CopyAssignable`. Inserts `n` copies of `t` before `p`.

§ 23.2.3 834

comparison function for `Key` is a refinement[257] of the partition into equivalent-key groups produced by `Pred`.

13 The iterator types `iterator` and `const_iterator` of an unordered associative container are of at least the forward iterator category. For unordered associative containers where the key type and value type are the same, both `iterator` and `const_iterator` are constant iterators.

14 The `insert` and `emplace` members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The `erase` members shall invalidate only iterators and references to the erased elements, and preserve the relative order of the elements that are not erased.

15 The `insert` and `emplace` members shall not affect the validity of iterators if `(N+n) <= z * B`, where `N` is the number of elements in the container prior to the insert operation, `n` is the number of elements inserted, `B` is the container's bucket count, and `z` is the container's maximum load factor.

16 The `extract` members invalidate only iterators to the removed element, and preserve the relative order of the elements that are not erased; pointers and references to the removed element remain valid. However, accessing the element through such pointers and references while the element is owned by a `node_type` is undefined behavior. References and pointers to an element obtained while it is owned by a `node_type` are invalidated if the element is successfully inserted.

23.2.7.1 Exception safety guarantees [unord.req.except]

1 For unordered associative containers, no `clear()` function throws an exception. `erase(k)` does not throw an exception unless that exception is thrown by the container's `Hash` or `Pred` object (if any).

2 For unordered associative containers, if an exception is thrown by any operation other than the container's hash function from within an `insert` or `emplace` function inserting a single element, the insertion has no effect.

3 For unordered associative containers, no `swap` function throws an exception unless that exception is thrown by the swap of the container's `Hash` or `Pred` object (if any).

4 For unordered associative containers, if an exception is thrown from within a `rehash()` function other than by the container's hash function or comparison function, the `rehash()` function has no effect.

23.3 Sequence containers [sequences]

23.3.1 In general [sequences.general]

1 The headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, and `<vector>` define class templates that meet the requirements for sequence containers.

23.3.2 Header <array> synopsis [array.syn]

```
#include <initializer_list>

namespace std {
  // 23.3.7, class template array
  template <class T, size_t N> struct array;
  template <class T, size_t N>
    bool operator==(const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
    bool operator!=(const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
    bool operator< (const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
    bool operator> (const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
```

257) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.

©ISO/IEC Dxxxx

comparison function for `Key` is a refinement[257] of the partition into equivalent-key groups produced by `Pred`.

13 The iterator types `iterator` and `const_iterator` of an unordered associative container are of at least the forward iterator category. For unordered associative containers where the key type and value type are the same, both `iterator` and `const_iterator` are constant iterators.

14 The `insert` and `emplace` members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The `erase` members shall invalidate only iterators and references to the erased elements, and preserve the relative order of the elements that are not erased.

15 The `insert` and `emplace` members shall not affect the validity of iterators if `(N+n) <= z * B`, where `N` is the number of elements in the container prior to the insert operation, `n` is the number of elements inserted, `B` is the container's bucket count, and `z` is the container's maximum load factor.

16 The `extract` members invalidate only iterators to the removed element, and preserve the relative order of the elements that are not erased; pointers and references to the removed element remain valid. However, accessing the element through such pointers and references while the element is owned by a `node_type` is undefined behavior. References and pointers to an element obtained while it is owned by a `node_type` are invalidated if the element is successfully inserted.

23.2.7.1 Exception safety guarantees [unord.req.except]

1 For unordered associative containers, no `clear()` function throws an exception. `erase(k)` does not throw an exception unless that exception is thrown by the container's `Hash` or `Pred` object (if any).

2 For unordered associative containers, if an exception is thrown by any operation other than the container's hash function from within an `insert` or `emplace` function inserting a single element, the insertion has no effect.

3 For unordered associative containers, no `swap` function throws an exception unless that exception is thrown by the swap of the container's `Hash` or `Pred` object (if any).

4 For unordered associative containers, if an exception is thrown from within a `rehash()` function other than by the container's hash function or comparison function, the `rehash()` function has no effect.

23.3 Sequence containers [sequences]

23.3.1 In general [sequences.general]

1 The headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, and `<vector>` define class templates that meet the requirements for sequence containers.

23.3.2 Header <array> synopsis [array.syn]

```
#include <initializer_list>

namespace std {
  // 23.3.7, class template array
  template <class T, size_t N> struct array;
  template <class T, size_t N>
    bool operator==(const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
    bool operator!=(const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
    bool operator< (const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
    bool operator> (const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
```

257) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.

§ 23.3.2 863

```
    bool operator<=(const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
    bool operator>=(const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
    void swap(array<T, N>& x, array<T, N>& y) noexcept(noexcept(x.swap(y)));

  template <class T> class tuple_size;
  template <size_t I, class T> class tuple_element;
  template <class T, size_t N>
    struct tuple_size<array<T, N>>;
  template <size_t I, class T, size_t N>
    struct tuple_element<I, array<T, N>>;
  template <size_t I, class T, size_t N>
    constexpr T& get(array<T, N>&) noexcept;
  template <size_t I, class T, size_t N>
    constexpr T&& get(array<T, N>&&) noexcept;
  template <size_t I, class T, size_t N>
    constexpr const T& get(const array<T, N>&) noexcept;
  template <size_t I, class T, size_t N>
    constexpr const T&& get(const array<T, N>&&) noexcept;
}
```

23.3.3 Header <deque> synopsis [deque.syn]

```
#include <initializer_list>

namespace std {
  // 23.3.8, class template deque
  template <class T, class Allocator = allocator<T>> class deque;
  template <class T, class Allocator>
    bool operator==(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    void swap(deque<T, Allocator>& x, deque<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  namespace pmr {
    template <class T>
      using deque = std::deque<T, polymorphic_allocator<T>>;
  }
}
```

23.3.4 Header <forward_list> synopsis [forward_list.syn]

```
#include <initializer_list>

namespace std {
  // 23.3.9, class template forward_list
```



```
    bool operator<=(const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
    bool operator>=(const array<T, N>& x, const array<T, N>& y);
  template <class T, size_t N>
    void swap(array<T, N>& x, array<T, N>& y) noexcept(noexcept(x.swap(y)));

  template <class T> class tuple_size;
  template <size_t I, class T> class tuple_element;
  template <class T, size_t N>
    struct tuple_size<array<T, N>>;
  template <size_t I, class T, size_t N>
    struct tuple_element<I, array<T, N>>;
  template <size_t I, class T, size_t N>
    constexpr T& get(array<T, N>&) noexcept;
  template <size_t I, class T, size_t N>
    constexpr T&& get(array<T, N>&&) noexcept;
  template <size_t I, class T, size_t N>
    constexpr const T& get(const array<T, N>&) noexcept;
  template <size_t I, class T, size_t N>
    constexpr const T&& get(const array<T, N>&&) noexcept;
}
```

23.3.3 Header <deque> synopsis [deque.syn]

```
#include <initializer_list>

namespace std {
  // 23.3.8, class template deque
  template <class T, class Allocator = allocator<T>> class deque;
  template <class T, class Allocator>
    bool operator==(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
  template <class T, class Allocator>
    void swap(deque<T, Allocator>& x, deque<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  namespace pmr {
    template <class T>
      using deque = std::deque<T, polymorphic_allocator<T>>;
  }
}
```

23.3.4 Header <forward_list> synopsis [forward_list.syn]

```
#include <initializer_list>

namespace std {
  // 23.3.9, class template forward_list
```

```
  template <class T, class Allocator = allocator<T>> class forward_list;
  template <class T, class Allocator>
    bool operator==(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    void swap(forward_list<T, Allocator>& x, forward_list<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  namespace pmr {
    template <class T>
      using forward_list = std::forward_list<T, polymorphic_allocator<T>>;
  }
}
```

23.3.5 Header <list> synopsis [list.syn]

```
#include <initializer_list>

namespace std {
  // 23.3.10, class template list
  template <class T, class Allocator = allocator<T>> class list;
  template <class T, class Allocator>
    bool operator==(const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    void swap(list<T, Allocator>& x, list<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  namespace pmr {
    template <class T>
      using list = std::list<T, polymorphic_allocator<T>>;
  }
}
```

23.3.6 Header <vector> synopsis [vector.syn]

```
#include <initializer_list>

namespace std {
```



```
  template <class T, class Allocator = allocator<T>> class forward_list;
  template <class T, class Allocator>
    bool operator==(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
  template <class T, class Allocator>
    void swap(forward_list<T, Allocator>& x, forward_list<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  namespace pmr {
    template <class T>
      using forward_list = std::forward_list<T, polymorphic_allocator<T>>;
  }
}
```

23.3.5 Header <list> synopsis [list.syn]

```
#include <initializer_list>

namespace std {
  // 23.3.10, class template list
  template <class T, class Allocator = allocator<T>> class list;
  template <class T, class Allocator>
    bool operator==(const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const list<T, Allocator>& x, const list<T, Allocator>& y);
  template <class T, class Allocator>
    void swap(list<T, Allocator>& x, list<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  namespace pmr {
    template <class T>
      using list = std::list<T, polymorphic_allocator<T>>;
  }
}
```

23.3.6 Header <vector> synopsis [vector.syn]

```
#include <initializer_list>

namespace std {
```

```
  // 23.3.11, class template vector
  template <class T, class Allocator = allocator<T>> class vector;
  template <class T, class Allocator>
    bool operator==(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  // 23.3.12, class vector<bool>
  template <class Allocator> class vector<bool, Allocator>;

  // hash support
  template <class T> struct hash;
  template <class Allocator> struct hash<vector<bool, Allocator>>;

  namespace pmr {
    template <class T>
      using vector = std::vector<T, polymorphic_allocator<T>>;
  }
}
```

23.3.7 Class template array [array]

23.3.7.1 Class template array overview [array.overview]

1 The header <array> defines a class template for storing fixed-size sequences of objects. An array is a contiguous container (23.2.1). An instance of array<T, N> stores N elements of type T, so that size() == N is an invariant.

2 An array is an aggregate (8.6.1) that can be list-initialized with up to N elements whose types are convertible to T.

3 An array satisfies all of the requirements of a container and of a reversible container (23.2), except that a default constructed array object is not empty and that swap does not have constant complexity. An array satisfies some of the requirements of a sequence container (23.2.3). Descriptions are provided here only for operations on array that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <class T, size_t N>
  struct array {
    // types:
    using value_type             = T;
    using pointer                = T*;
    using const_pointer          = const T*;
    using reference              = T&;
    using const_reference        = const T&;
    using size_type              = size_t;
```



```
  // 23.3.11, class template vector
  template <class T, class Allocator = allocator<T>> class vector;
  template <class T, class Allocator>
    bool operator==(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator< (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator!=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator> (const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator>=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    bool operator<=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
  template <class T, class Allocator>
    void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  // 23.3.12, class vector<bool>
  template <class Allocator> class vector<bool, Allocator>;

  // hash support
  template <class T> struct hash;
  template <class Allocator> struct hash<vector<bool, Allocator>>;

  namespace pmr {
    template <class T>
      using vector = std::vector<T, polymorphic_allocator<T>>;
  }
}
```

23.3.7 Class template array [array]

23.3.7.1 Class template array overview [array.overview]

1 The header <array> defines a class template for storing fixed-size sequences of objects. An array is a contiguous container (23.2.1). An instance of array<T, N> stores N elements of type T, so that size() == N is an invariant.

2 An array is an aggregate (8.6.1) that can be list-initialized with up to N elements whose types are convertible to T.

3 An array satisfies all of the requirements of a container and of a reversible container (23.2), except that a default constructed array object is not empty and that swap does not have constant complexity. An array satisfies some of the requirements of a sequence container (23.2.3). Descriptions are provided here only for operations on array that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <class T, size_t N>
  struct array {
    // types:
    using value_type             = T;
    using pointer                = T*;
    using const_pointer          = const T*;
    using reference              = T&;
    using const_reference        = const T&;
    using size_type              = size_t;
```

```
  template <class... Args> iterator emplace_after(const_iterator position, Args&&... args);
  iterator insert_after(const_iterator position, const T& x);
  iterator insert_after(const_iterator position, T&& x);

  iterator insert_after(const_iterator position, size_type n, const T& x);
  template <class InputIterator>
    iterator insert_after(const_iterator position, InputIterator first, InputIterator last);
  iterator insert_after(const_iterator position, initializer_list<T> il);

  iterator erase_after(const_iterator position);
  iterator erase_after(const_iterator position, const_iterator last);
  void swap(forward_list&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value);

  void resize(size_type sz);
  void resize(size_type sz, const value_type& c);
  void clear() noexcept;

  // 23.3.9.6, forward_list operations
  void splice_after(const_iterator position, forward_list& x);
  void splice_after(const_iterator position, forward_list&& x);
  void splice_after(const_iterator position, forward_list& x,
                    const_iterator i);
  void splice_after(const_iterator position, forward_list&& x,
                    const_iterator i);
  void splice_after(const_iterator position, forward_list& x,
                    const_iterator first, const_iterator last);
  void splice_after(const_iterator position, forward_list&& x,
                    const_iterator first, const_iterator last);

  void remove(const T& value);
  template <class Predicate> void remove_if(Predicate pred);

  void unique();
  template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);

  void merge(forward_list& x);
  void merge(forward_list&& x);
  template <class Compare> void merge(forward_list& x, Compare comp);
  template <class Compare> void merge(forward_list&& x, Compare comp);

  void sort();
  template <class Compare> void sort(Compare comp);

  void reverse() noexcept;
};

template <class T, class Allocator>
  bool operator==(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
template <class T, class Allocator>
  bool operator< (const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
template <class T, class Allocator>
  bool operator!=(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
template <class T, class Allocator>
```



```
  template <class... Args> iterator emplace_after(const_iterator position, Args&&... args);
  iterator insert_after(const_iterator position, const T& x);
  iterator insert_after(const_iterator position, T&& x);

  iterator insert_after(const_iterator position, size_type n, const T& x);
  template <class InputIterator>
    iterator insert_after(const_iterator position, InputIterator first, InputIterator last);
  iterator insert_after(const_iterator position, initializer_list<T> il);

  iterator erase_after(const_iterator position);
  iterator erase_after(const_iterator position, const_iterator last);
  void swap(forward_list&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value);

  void resize(size_type sz);
  void resize(size_type sz, const value_type& c);
  void clear() noexcept;

  // 23.3.9.6, forward_list operations
  void splice_after(const_iterator position, forward_list& x);
  void splice_after(const_iterator position, forward_list&& x);
  void splice_after(const_iterator position, forward_list& x,
                    const_iterator i);
  void splice_after(const_iterator position, forward_list&& x,
                    const_iterator i);
  void splice_after(const_iterator position, forward_list& x,
                    const_iterator first, const_iterator last);
  void splice_after(const_iterator position, forward_list&& x,
                    const_iterator first, const_iterator last);

  void remove(const T& value);
  template <class Predicate> void remove_if(Predicate pred);

  void unique();
  template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);

  void merge(forward_list& x);
  void merge(forward_list&& x);
  template <class Compare> void merge(forward_list& x, Compare comp);
  template <class Compare> void merge(forward_list&& x, Compare comp);

  void sort();
  template <class Compare> void sort(Compare comp);

  void reverse() noexcept;
};

template <class T, class Allocator>
  bool operator==(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
template <class T, class Allocator>
  bool operator< (const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
template <class T, class Allocator>
  bool operator!=(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);
template <class T, class Allocator>
```

```
    iterator insert(const_iterator position, const bool& x);
    iterator insert(const_iterator position, size_type n, const bool& x);
    template <class InputIterator>
      iterator insert(const_iterator position,
                      InputIterator first, InputIterator last);
    iterator insert(const_iterator position, initializer_list<bool> il);

    iterator erase(const_iterator position);
    iterator erase(const_iterator first, const_iterator last);
    void swap(vector<bool, Allocator>&);
    static void swap(reference x, reference y) noexcept;
    void flip() noexcept;        // flips all bits
    void clear() noexcept;
  };
}
```

2 Unless described below, all operations have the same requirements and semantics as the primary `vector` template, except that operations dealing with the `bool` value type map to bit values in the container storage and `allocator_traits::construct` (20.10.8.2) is not used to construct these values.

3 There is no requirement that the data be stored as a contiguous allocation of `bool` values. A space-optimized representation of bits is recommended instead.

4 `reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`. The conversion function returns `true` when the bit is set, and `false` otherwise. The assignment operator sets the bit when the argument is (convertible to) `true` and clears it otherwise. `flip` reverses the state of the bit.

```
void flip() noexcept;
```

5 *Effects:* Replaces each element in the container with its complement.

```
static void swap(reference x, reference y) noexcept;
```

6 *Effects:* Exchanges the contents of `x` and `y` as if by:

```
bool b = x;
x = y;
y = b;
```

```
template <class Allocator> struct hash<vector<bool, Allocator>>;
```

7 The specialization is enabled (20.14.15).

23.4 Associative containers [associative]

23.4.1 In general [associative.general]

1 The header `<map>` defines the class templates `map` and `multimap`; the header `<set>` defines the class templates `set` and `multiset`.

23.4.2 Header `<map>` synopsis [associative.map.syn]

```
#include <initializer_list>

namespace std {
  // 23.4.4, class template map
  template <class Key, class T, class Compare = default_order_t<Key>,
            class Allocator = allocator<pair<const Key, T>>>
    class map;
  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const map<Key, T, Compare, Allocator>& x,
```



```
    iterator insert(const_iterator position, const bool& x);
    iterator insert(const_iterator position, size_type n, const bool& x);
    template <class InputIterator>
      iterator insert(const_iterator position,
                      InputIterator first, InputIterator last);
    iterator insert(const_iterator position, initializer_list<bool> il);

    iterator erase(const_iterator position);
    iterator erase(const_iterator first, const_iterator last);
    void swap(vector<bool, Allocator>&);
    static void swap(reference x, reference y) noexcept;
    void flip() noexcept;        // flips all bits
    void clear() noexcept;
  };
}
```

2 Unless described below, all operations have the same requirements and semantics as the primary `vector` template, except that operations dealing with the `bool` value type map to bit values in the container storage and `allocator_traits::construct` (20.10.8.2) is not used to construct these values.

3 There is no requirement that the data be stored as a contiguous allocation of `bool` values. A space-optimized representation of bits is recommended instead.

4 `reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`. The conversion function returns `true` when the bit is set, and `false` otherwise. The assignment operator sets the bit when the argument is (convertible to) `true` and clears it otherwise. `flip` reverses the state of the bit.

```
void flip() noexcept;
```

5 *Effects:* Replaces each element in the container with its complement.

```
static void swap(reference x, reference y) noexcept;
```

6 *Effects:* Exchanges the contents of `x` and `y` as if by:

```
bool b = x;
x = y;
y = b;
```

```
template <class Allocator> struct hash<vector<bool, Allocator>>;
```

7 The specialization is enabled (20.14.15).

23.4 Associative containers [associative]

23.4.1 In general [associative.general]

1 The header `<map>` defines the class templates `map` and `multimap`; the header `<set>` defines the class templates `set` and `multiset`.

23.4.2 Header `<map>` synopsis [associative.map.syn]

```
#include <initializer_list>

namespace std {
  // 23.4.4, class template map
  template <class Key, class T, class Compare = default_order_t<Key>,
            class Allocator = allocator<pair<const Key, T>>>
    class map;
  template <class Key, class T, class Compare, class Allocator>
    bool operator==(const map<Key, T, Compare, Allocator>& x,
```

```
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator< (const map<Key, T, Compare, Allocator>& x,
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator!=(const map<Key, T, Compare, Allocator>& x,
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator> (const map<Key, T, Compare, Allocator>& x,
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator>=(const map<Key, T, Compare, Allocator>& x,
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator<=(const map<Key, T, Compare, Allocator>& x,
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  void swap(map<Key, T, Compare, Allocator>& x,
            map<Key, T, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));

// 23.4.5, class template multimap
template <class Key, class T, class Compare = default_order_t<Key>,
          class Allocator = allocator<pair<const Key, T>>>
  class multimap;
template <class Key, class T, class Compare, class Allocator>
  bool operator==(const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator< (const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator!=(const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator> (const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator>=(const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator<=(const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  void swap(multimap<Key, T, Compare, Allocator>& x,
            multimap<Key, T, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));

namespace pmr {
  template <class Key, class T, class Compare = default_order_t<Key>>
    using map = std::map<Key, T, Compare,
                         polymorphic_allocator<pair<const Key, T>>>;

  template <class Key, class T, class Compare = default_order_t<Key>>
    using multimap = std::multimap<Key, T, Compare,
```



```
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator< (const map<Key, T, Compare, Allocator>& x,
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator!=(const map<Key, T, Compare, Allocator>& x,
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator> (const map<Key, T, Compare, Allocator>& x,
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator>=(const map<Key, T, Compare, Allocator>& x,
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator<=(const map<Key, T, Compare, Allocator>& x,
                  const map<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  void swap(map<Key, T, Compare, Allocator>& x,
            map<Key, T, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));

// 23.4.5, class template multimap
template <class Key, class T, class Compare = default_order_t<Key>,
          class Allocator = allocator<pair<const Key, T>>>
  class multimap;
template <class Key, class T, class Compare, class Allocator>
  bool operator==(const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator< (const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator!=(const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator> (const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator>=(const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  bool operator<=(const multimap<Key, T, Compare, Allocator>& x,
                  const multimap<Key, T, Compare, Allocator>& y);
template <class Key, class T, class Compare, class Allocator>
  void swap(multimap<Key, T, Compare, Allocator>& x,
            multimap<Key, T, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));

namespace pmr {
  template <class Key, class T, class Compare = default_order_t<Key>>
    using map = std::map<Key, T, Compare,
                         polymorphic_allocator<pair<const Key, T>>>;

  template <class Key, class T, class Compare = default_order_t<Key>>
    using multimap = std::multimap<Key, T, Compare,
```

```
                                polymorphic_allocator<pair<const Key, T>>>;
  }
}
```

23.4.3 Header <set> synopsis [associative.set.syn]

```
#include <initializer_list>

namespace std {
  // 23.4.6, class template set
  template <class Key, class Compare = default_order_t<Key>,
            class Allocator = allocator<Key>>
    class set;
  template <class Key, class Compare, class Allocator>
    bool operator==(const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator!=(const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator> (const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator>=(const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator<=(const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    void swap(set<Key, Compare, Allocator>& x,
              set<Key, Compare, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  // 23.4.6, class template multiset
  template <class Key, class Compare = default_order_t<Key>,
            class Allocator = allocator<Key>>
    class multiset;
  template <class Key, class Compare, class Allocator>
    bool operator==(const multiset<Key, Compare, Allocator>& x,
                    const multiset<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const multiset<Key, Compare, Allocator>& x,
                    const multiset<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator!=(const multiset<Key, Compare, Allocator>& x,
                    const multiset<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator> (const multiset<Key, Compare, Allocator>& x,
                    const multiset<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator>=(const multiset<Key, Compare, Allocator>& x,
                    const multiset<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
```



```
                                polymorphic_allocator<pair<const Key, T>>>;
  }
}
```

23.4.3 Header <set> synopsis [associative.set.syn]

```
#include <initializer_list>

namespace std {
  // 23.4.6, class template set
  template <class Key, class Compare = default_order_t<Key>,
            class Allocator = allocator<Key>>
    class set;
  template <class Key, class Compare, class Allocator>
    bool operator==(const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator!=(const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator> (const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator>=(const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator<=(const set<Key, Compare, Allocator>& x,
                    const set<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    void swap(set<Key, Compare, Allocator>& x,
              set<Key, Compare, Allocator>& y)
      noexcept(noexcept(x.swap(y)));

  // 23.4.6, class template multiset
  template <class Key, class Compare = default_order_t<Key>,
            class Allocator = allocator<Key>>
    class multiset;
  template <class Key, class Compare, class Allocator>
    bool operator==(const multiset<Key, Compare, Allocator>& x,
                    const multiset<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator< (const multiset<Key, Compare, Allocator>& x,
                    const multiset<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator!=(const multiset<Key, Compare, Allocator>& x,
                    const multiset<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator> (const multiset<Key, Compare, Allocator>& x,
                    const multiset<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
    bool operator>=(const multiset<Key, Compare, Allocator>& x,
                    const multiset<Key, Compare, Allocator>& y);
  template <class Key, class Compare, class Allocator>
```

```
      noexcept(noexcept(x.swap(y)));
}
```

23.4.7.2 multiset constructors [multiset.cons]

```
explicit multiset(const Compare& comp, const Allocator& = Allocator());
```

1 *Effects:* Constructs an empty `multiset` using the specified comparison object and allocator.

2 *Complexity:* Constant.

```
template <class InputIterator>
  multiset(InputIterator first, InputIterator last,
           const Compare& comp = Compare(), const Allocator& = Allocator());
```

3 *Effects:* Constructs an empty `multiset` using the specified comparison object and allocator, and inserts elements from the range `[first, last)`.

4 *Complexity:* Linear in N if the range `[first, last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

23.4.7.3 multiset specialized algorithms [multiset.special]

```
template <class Key, class Compare, class Allocator>
  void swap(multiset<Key, Compare, Allocator>& x,
            multiset<Key, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));
```

1 *Effects:* As if by `x.swap(y)`.

23.5 Unordered associative containers [unord]

23.5.1 In general [unord.general]

1 The header `<unordered_map>` defines the class templates `unordered_map` and `unordered_multimap`; the header `<unordered_set>` defines the class templates `unordered_set` and `unordered_multiset`.

23.5.2 Header <unordered_map> synopsis [unord.map.syn]

```
#include <initializer_list>

namespace std {
  // 23.5.4, class template unordered_map
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>,
            class Alloc = allocator<pair<const Key, T>>>
    class unordered_map;

  // 23.5.5, class template unordered_multimap
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>,
            class Alloc = allocator<pair<const Key, T>>>
    class unordered_multimap;

  template <class Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
```



```
      noexcept(noexcept(x.swap(y)));
}
```

23.4.7.2 multiset constructors [multiset.cons]

```
explicit multiset(const Compare& comp, const Allocator& = Allocator());
```

1 *Effects:* Constructs an empty `multiset` using the specified comparison object and allocator.

2 *Complexity:* Constant.

```
template <class InputIterator>
  multiset(InputIterator first, InputIterator last,
           const Compare& comp = Compare(), const Allocator& = Allocator());
```

3 *Effects:* Constructs an empty `multiset` using the specified comparison object and allocator, and inserts elements from the range `[first, last)`.

4 *Complexity:* Linear in N if the range `[first, last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

23.4.7.3 multiset specialized algorithms [multiset.special]

```
template <class Key, class Compare, class Allocator>
  void swap(multiset<Key, Compare, Allocator>& x,
            multiset<Key, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));
```

1 *Effects:* As if by `x.swap(y)`.

23.5 Unordered associative containers [unord]

23.5.1 In general [unord.general]

1 The header `<unordered_map>` defines the class templates `unordered_map` and `unordered_multimap`; the header `<unordered_set>` defines the class templates `unordered_set` and `unordered_multiset`.

23.5.2 Header <unordered_map> synopsis [unord.map.syn]

```
#include <initializer_list>

namespace std {
  // 23.5.4, class template unordered_map
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>,
            class Alloc = allocator<pair<const Key, T>>>
    class unordered_map;

  // 23.5.5, class template unordered_multimap
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>,
            class Alloc = allocator<pair<const Key, T>>>
    class unordered_multimap;

  template <class Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
```

```
            unordered_map<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

  template <class Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);

  namespace pmr {
    template <class Key,
              class T,
              class Hash = hash<Key>,
              class Pred = equal_to<Key>>
      using unordered_map =
        std::unordered_map<Key, T, Hash, Pred,
                           polymorphic_allocator<pair<const Key, T>>>;
    template <class Key,
              class T,
              class Hash = hash<Key>,
              class Pred = equal_to<Key>>
      using unordered_multimap =
        std::unordered_multimap<Key, T, Hash, Pred,
                                polymorphic_allocator<pair<const Key, T>>>;

  }
}
```

23.5.3 Header <unordered_set> synopsis [unord.set.syn]

```
#include <initializer_list>

namespace std {
  // 23.5.6, class template unordered_set
  template <class Key,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>,
            class Alloc = allocator<Key>>
    class unordered_set;

  // 23.5.7, class template unordered_multiset
  template <class Key,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>,
```



```
            unordered_map<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

  template <class Key, class T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));

  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);
  template <class Key, class T, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_multimap<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_multimap<Key, T, Hash, Pred, Alloc>& b);

  namespace pmr {
    template <class Key,
              class T,
              class Hash = hash<Key>,
              class Pred = equal_to<Key>>
      using unordered_map =
        std::unordered_map<Key, T, Hash, Pred,
                           polymorphic_allocator<pair<const Key, T>>>;
    template <class Key,
              class T,
              class Hash = hash<Key>,
              class Pred = equal_to<Key>>
      using unordered_multimap =
        std::unordered_multimap<Key, T, Hash, Pred,
                                polymorphic_allocator<pair<const Key, T>>>;

  }
}
```

23.5.3 Header <unordered_set> synopsis [unord.set.syn]

```
#include <initializer_list>

namespace std {
  // 23.5.6, class template unordered_set
  template <class Key,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>,
            class Alloc = allocator<Key>>
    class unordered_set;

  // 23.5.7, class template unordered_multiset
  template <class Key,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>,
```

(2.2) — Operations on those sequence elements that are required by its specification.

(2.3) — User-provided function objects to be applied during the execution of the algorithm, if required by the specification.

(2.4) — Operations on those function objects required by the specification. [*Note:* See 25.1. — *end note*]

These functions are herein called *element access functions*. [*Example:* The `sort` function may invoke the following element access functions:

(2.5) — Operations of the random-access iterator of the actual template argument (as per 24.2.7), as implied by the name of the template parameter `RandomAccessIterator`.

(2.6) — The `swap` function on the elements of the sequence (as per the preconditions specified in 25.7.1.1).

(2.7) — The user-provided `Compare` function object.

— *end example*]

25.4.2 Requirements on user-provided function objects [algorithms.parallel.user]

1 Function objects passed into parallel algorithms as objects of type `Predicate`, `BinaryPredicate`, `Compare`, and `BinaryOperation` shall not directly or indirectly modify objects via their arguments.

25.4.3 Effect of execution policies on algorithm execution [algorithms.parallel.exec]

1 Parallel algorithms have template parameters named `ExecutionPolicy` (20.19) which describe the manner in which the execution of these algorithms may be parallelized and the manner in which they apply the element access functions.

2 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::sequenced_policy` all occur in the calling thread of execution. [*Note:* The invocations are not interleaved; see 1.9. — *end note*]

3 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::parallel_policy` are permitted to execute in either the invoking thread of execution or in a thread of execution implicitly created by the library to support parallel algorithm execution. If the threads of execution created by `thread` (30.3.2) provide concurrent forward progress guarantees (1.10.2), then a thread of execution implicitly created by the library will provide parallel forward progress guarantees; otherwise, the provided forward progress guarantee is implementation-defined. Any such invocations executing in the same thread of execution are indeterminately sequenced with respect to each other. [*Note:* It is the caller's responsibility to ensure that the invocation does not introduce data races or deadlocks. — *end note*]
[*Example:*

```
int a[] = {0,1};
std::vector<int> v;
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int i) {
  v.push_back(i*2+1); // incorrect: data race
});
```

The program above has a data race because of the unsynchronized access to the container `v`. — *end example*]
[*Example:*

```
std::atomic<int> x{0};
int a[] = {1,2};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
  x.fetch_add(1, std::memory_order_relaxed);
  // spin wait for another iteration to change the value of x
  while (x.load(std::memory_order_relaxed) == 1) { } // incorrect: assumes execution order
});
```


©ISO/IEC Dxxxx

(2.2) — Operations on those sequence elements that are required by its specification.

(2.3) — User-provided function objects to be applied during the execution of the algorithm, if required by the specification.

(2.4) — Operations on those function objects required by the specification. [*Note:* See 25.1. — *end note*]

These functions are herein called *element access functions*. [*Example:* The `sort` function may invoke the following element access functions:

(2.5) — Operations of the random-access iterator of the actual template argument (as per 24.2.7), as implied by the name of the template parameter `RandomAccessIterator`.

(2.6) — The `swap` function on the elements of the sequence (as per the preconditions specified in 25.7.1.1).

(2.7) — The user-provided `Compare` function object.

— *end example*]

25.4.2 Requirements on user-provided function objects [algorithms.parallel.user]

1 Function objects passed into parallel algorithms as objects of type `Predicate`, `BinaryPredicate`, `Compare`, and `BinaryOperation` shall not directly or indirectly modify objects via their arguments.

25.4.3 Effect of execution policies on algorithm execution [algorithms.parallel.exec]

1 Parallel algorithms have template parameters named `ExecutionPolicy` (20.19) which describe the manner in which the execution of these algorithms may be parallelized and the manner in which they apply the element access functions.

2 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::sequenced_policy` all occur in the calling thread of execution. [*Note:* The invocations are not interleaved; see 1.9. — *end note*]

3 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::parallel_policy` are permitted to execute in either the invoking thread of execution or in a thread of execution implicitly created by the library to support parallel algorithm execution. If the threads of execution created by `thread` (30.3.2) provide concurrent forward progress guarantees (1.10.2), then a thread of execution implicitly created by the library will provide parallel forward progress guarantees; otherwise, the provided forward progress guarantee is implementation-defined. Any such invocations executing in the same thread of execution are indeterminately sequenced with respect to each other. [*Note:* It is the caller's responsibility to ensure that the invocation does not introduce data races or deadlocks. — *end note*]
[*Example:*

```
int a[] = {0,1};
std::vector<int> v;
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int i) {
  v.push_back(i*2+1); // incorrect: data race
});
```

The program above has a data race because of the unsynchronized access to the container `v`. — *end example*]
[*Example:*

```
std::atomic<int> x{0};
int a[] = {1,2};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
  x.fetch_add(1, std::memory_order_relaxed);
  // spin wait for another iteration to change the value of x
  while (x.load(std::memory_order_relaxed) == 1) { } // incorrect: assumes execution order
});
```

§ 25.4.3 998

have a corresponding constructor subject to the same requirements and taking arguments identical in number, type, and default values. Moreover, for each of the member functions of `D` that return values corresponding to parameters of the distribution, `P` shall have a corresponding member function with the identical name, type, and semantics.

10 `P` shall have a declaration of the form

```
using distribution_type =  D;
```

26.6.2 Header <random> synopsis [rand.synopsis]

```
#include <initializer_list>

namespace std {
  // 26.6.3.1, class template linear_congruential_engine
  template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

  // 26.6.3.2, class template mersenne_twister_engine
  template<class UIntType, size_t w, size_t n, size_t m, size_t r,
           UIntType a, size_t u, UIntType d, size_t s,
           UIntType b, size_t t,
           UIntType c, size_t l, UIntType f>
    class mersenne_twister_engine;

  // 26.6.3.3, class template subtract_with_carry_engine
  template<class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine;

  // 26.6.4.2, class template discard_block_engine
  template<class Engine, size_t p, size_t r>
    class discard_block_engine;

  // 26.6.4.3, class template independent_bits_engine
  template<class Engine, size_t w, class UIntType>
    class independent_bits_engine;

  // 26.6.4.4, class template shuffle_order_engine
  template<class Engine, size_t k>
    class shuffle_order_engine;

  // 26.6.5, engines and engine adaptors with predefined parameters
  using minstd_rand0  = see below;
  using minstd_rand   = see below;
  using mt19937       = see below;
  using mt19937_64    = see below;
  using ranlux24_base = see below;
  using ranlux48_base = see below;
  using ranlux24      = see below;
  using ranlux48      = see below;
  using knuth_b       = see below;

  using default_random_engine = see below;

  // 26.6.6, class random_device
  class random_device;
```


©ISO/IEC Dxxxx

have a corresponding constructor subject to the same requirements and taking arguments identical in number, type, and default values. Moreover, for each of the member functions of `D` that return values corresponding to parameters of the distribution, `P` shall have a corresponding member function with the identical name, type, and semantics.

10 `P` shall have a declaration of the form

```
using distribution_type =  D;
```

26.6.2 Header <random> synopsis [rand.synopsis]

```
#include <initializer_list>

namespace std {
  // 26.6.3.1, class template linear_congruential_engine
  template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

  // 26.6.3.2, class template mersenne_twister_engine
  template<class UIntType, size_t w, size_t n, size_t m, size_t r,
           UIntType a, size_t u, UIntType d, size_t s,
           UIntType b, size_t t,
           UIntType c, size_t l, UIntType f>
    class mersenne_twister_engine;

  // 26.6.3.3, class template subtract_with_carry_engine
  template<class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine;

  // 26.6.4.2, class template discard_block_engine
  template<class Engine, size_t p, size_t r>
    class discard_block_engine;

  // 26.6.4.3, class template independent_bits_engine
  template<class Engine, size_t w, class UIntType>
    class independent_bits_engine;

  // 26.6.4.4, class template shuffle_order_engine
  template<class Engine, size_t k>
    class shuffle_order_engine;

  // 26.6.5, engines and engine adaptors with predefined parameters
  using minstd_rand0  = see below;
  using minstd_rand   = see below;
  using mt19937       = see below;
  using mt19937_64    = see below;
  using ranlux24_base = see below;
  using ranlux48_base = see below;
  using ranlux24      = see below;
  using ranlux48      = see below;
  using knuth_b       = see below;

  using default_random_engine = see below;

  // 26.6.6, class random_device
  class random_device;
```

§ 26.6.2 1059

```
// 26.6.7.1, class seed_seq
class seed_seq;

// 26.6.7.2, function template generate_canonical
template<class RealType, size_t bits, class URBG>
  RealType generate_canonical(URBG& g);

// 26.6.8.2.1, class template uniform_int_distribution
template<class IntType = int>
  class uniform_int_distribution;

// 26.6.8.2.2, class template uniform_real_distribution
template<class RealType = double>
  class uniform_real_distribution;

// 26.6.8.3.1, class bernoulli_distribution
class bernoulli_distribution;

// 26.6.8.3.2, class template binomial_distribution
template<class IntType = int>
  class binomial_distribution;

// 26.6.8.3.3, class template geometric_distribution
template<class IntType = int>
  class geometric_distribution;

// 26.6.8.3.4, class template negative_binomial_distribution
template<class IntType = int>
  class negative_binomial_distribution;

// 26.6.8.4.1, class template poisson_distribution
template<class IntType = int>
  class poisson_distribution;

// 26.6.8.4.2, class template exponential_distribution
template<class RealType = double>
  class exponential_distribution;

// 26.6.8.4.3, class template gamma_distribution
template<class RealType = double>
  class gamma_distribution;

// 26.6.8.4.4, class template weibull_distribution
template<class RealType = double>
  class weibull_distribution;

// 26.6.8.4.5, class template extreme_value_distribution
template<class RealType = double>
  class extreme_value_distribution;

// 26.6.8.5.1, class template normal_distribution
template<class RealType = double>
  class normal_distribution;

// 26.6.8.5.2, class template lognormal_distribution
```



```
// 26.6.7.1, class seed_seq
class seed_seq;

// 26.6.7.2, function template generate_canonical
template<class RealType, size_t bits, class URBG>
  RealType generate_canonical(URBG& g);

// 26.6.8.2.1, class template uniform_int_distribution
template<class IntType = int>
  class uniform_int_distribution;

// 26.6.8.2.2, class template uniform_real_distribution
template<class RealType = double>
  class uniform_real_distribution;

// 26.6.8.3.1, class bernoulli_distribution
class bernoulli_distribution;

// 26.6.8.3.2, class template binomial_distribution
template<class IntType = int>
  class binomial_distribution;

// 26.6.8.3.3, class template geometric_distribution
template<class IntType = int>
  class geometric_distribution;

// 26.6.8.3.4, class template negative_binomial_distribution
template<class IntType = int>
  class negative_binomial_distribution;

// 26.6.8.4.1, class template poisson_distribution
template<class IntType = int>
  class poisson_distribution;

// 26.6.8.4.2, class template exponential_distribution
template<class RealType = double>
  class exponential_distribution;

// 26.6.8.4.3, class template gamma_distribution
template<class RealType = double>
  class gamma_distribution;

// 26.6.8.4.4, class template weibull_distribution
template<class RealType = double>
  class weibull_distribution;

// 26.6.8.4.5, class template extreme_value_distribution
template<class RealType = double>
  class extreme_value_distribution;

// 26.6.8.5.1, class template normal_distribution
template<class RealType = double>
  class normal_distribution;

// 26.6.8.5.2, class template lognormal_distribution
```

```
  template<class RealType = double>
    class lognormal_distribution;

  // 26.6.8.5.3, class template chi_squared_distribution
  template<class RealType = double>
    class chi_squared_distribution;

  // 26.6.8.5.4, class template cauchy_distribution
  template<class RealType = double>
    class cauchy_distribution;

  // 26.6.8.5.5, class template fisher_f_distribution
  template<class RealType = double>
    class fisher_f_distribution;

  // 26.6.8.5.6, class template student_t_distribution
  template<class RealType = double>
    class student_t_distribution;

  // 26.6.8.6.1, class template discrete_distribution
  template<class IntType = int>
    class discrete_distribution;

  // 26.6.8.6.2, class template piecewise_constant_distribution
  template<class RealType = double>
    class piecewise_constant_distribution;

  // 26.6.8.6.3, class template piecewise_linear_distribution
  template<class RealType = double>
    class piecewise_linear_distribution;
}
```

26.6.3 Random number engine class templates [rand.eng]

1 Each type instantiated from a class template specified in this section 26.6.3 satisfies the requirements of a random number engine (26.6.1.4) type.

2 Except where specified otherwise, the complexity of each function specified in this section 26.6.3 is constant.

3 Except where specified otherwise, no function described in this section 26.6.3 throws an exception.

4 Every function described in this section 26.6.3 that has a function parameter `q` of type `Sseq&` for a template type parameter named `Sseq` that is different from type `seed_seq` throws what and when the invocation of `q.generate` throws.

5 Descriptions are provided in this section 26.6.3 only for engine operations that are not described in 26.6.1.4 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

6 Each template specified in this section 26.6.3 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

7 For every random number engine and for every random number engine adaptor `X` defined in this subclause (26.6.3) and in subclause 26.6.4:

(7.1) — if the constructor

©ISO/IEC Dxxxx

```
  template<class RealType = double>
    class lognormal_distribution;

  // 26.6.8.5.3, class template chi_squared_distribution
  template<class RealType = double>
    class chi_squared_distribution;

  // 26.6.8.5.4, class template cauchy_distribution
  template<class RealType = double>
    class cauchy_distribution;

  // 26.6.8.5.5, class template fisher_f_distribution
  template<class RealType = double>
    class fisher_f_distribution;

  // 26.6.8.5.6, class template student_t_distribution
  template<class RealType = double>
    class student_t_distribution;

  // 26.6.8.6.1, class template discrete_distribution
  template<class IntType = int>
    class discrete_distribution;

  // 26.6.8.6.2, class template piecewise_constant_distribution
  template<class RealType = double>
    class piecewise_constant_distribution;

  // 26.6.8.6.3, class template piecewise_linear_distribution
  template<class RealType = double>
    class piecewise_linear_distribution;
}
```

26.6.3 Random number engine class templates [rand.eng]

1 Each type instantiated from a class template specified in this section 26.6.3 satisfies the requirements of a random number engine (26.6.1.4) type.

2 Except where specified otherwise, the complexity of each function specified in this section 26.6.3 is constant.

3 Except where specified otherwise, no function described in this section 26.6.3 throws an exception.

4 Every function described in this section 26.6.3 that has a function parameter `q` of type `Sseq&` for a template type parameter named `Sseq` that is different from type `seed_seq` throws what and when the invocation of `q.generate` throws.

5 Descriptions are provided in this section 26.6.3 only for engine operations that are not described in 26.6.1.4 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

6 Each template specified in this section 26.6.3 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

7 For every random number engine and for every random number engine adaptor `X` defined in this subclause (26.6.3) and in subclause 26.6.4:

(7.1) — if the constructor

§ 26.6.3 1061

27.10.4.19 **[fs.def.relative-path]**

relative path

A path that is not absolute, and as such, only unambiguously identifies the location of a file when resolved (27.10.4.18) relative to an implied starting location. The elements of a path that determine if it is relative are operating system dependent. [*Note:* Pathnames "." and ".." are relative paths. — *end note*]

27.10.4.20 **[fs.def.symlink]**

symbolic link

A type of file with the property that when the file is encountered during pathname resolution, a string stored by the file is used to modify the pathname resolution. [*Note:* Symbolic links are often called symlinks. A symbolic link can be thought of as a raw pointer to a file. If the file pointed to does not exist, the symbolic link is said to be a "dangling" symbolic link. — *end note*]

27.10.5 Requirements [fs.req]

1 Throughout this subclause, `char`, `wchar_t`, `char16_t`, and `char32_t` are collectively called *encoded character types*.

2 Functions with template parameters named `EcharT` shall not participate in overload resolution unless `EcharT` is one of the encoded character types.

3 Template parameters named `InputIterator` shall meet the input iterator requirements (24.2.3) and shall have a value type that is one of the encoded character types.

4 [*Note:* Use of an encoded character type implies an associated encoding. Since `signed char` and `unsigned char` have no implied encoding, they are not included as permitted types. — *end note*]

5 Template parameters named `Allocator` shall meet the Allocator requirements (17.5.3.5).

27.10.5.1 Namespaces and headers [fs.req.namespace]

1 Unless otherwise specified, references to entities described in this subclause are assumed to be qualified with `::std::filesystem::`.

27.10.6 Header `<filesystem>` synopsis [fs.filesystem.syn]

```
namespace std::filesystem {
  // 27.10.8, paths
  class path;

  // 27.10.8.6, path non-member functions
  void swap(path& lhs, path& rhs) noexcept;
  size_t hash_value(const path& p) noexcept;

  bool operator==(const path& lhs, const path& rhs) noexcept;
  bool operator!=(const path& lhs, const path& rhs) noexcept;
  bool operator< (const path& lhs, const path& rhs) noexcept;
  bool operator<=(const path& lhs, const path& rhs) noexcept;
  bool operator> (const path& lhs, const path& rhs) noexcept;
  bool operator>=(const path& lhs, const path& rhs) noexcept;

  path operator/ (const path& lhs, const path& rhs);

  // 27.10.8.6.1, path inserter and extractor
  template <class charT, class traits>
    basic_ostream<charT, traits>&
      operator<<(basic_ostream<charT, traits>& os, const path& p);
  template <class charT, class traits>
```


©ISO/IEC Dxxxx

27.10.4.19 **[fs.def.relative-path]**

relative path

A path that is not absolute, and as such, only unambiguously identifies the location of a file when resolved (27.10.4.18) relative to an implied starting location. The elements of a path that determine if it is relative are operating system dependent. [*Note:* Pathnames "." and ".." are relative paths. — *end note*]

27.10.4.20 **[fs.def.symlink]**

symbolic link

A type of file with the property that when the file is encountered during pathname resolution, a string stored by the file is used to modify the pathname resolution. [*Note:* Symbolic links are often called symlinks. A symbolic link can be thought of as a raw pointer to a file. If the file pointed to does not exist, the symbolic link is said to be a "dangling" symbolic link. — *end note*]

27.10.5 Requirements [fs.req]

1 Throughout this subclause, `char`, `wchar_t`, `char16_t`, and `char32_t` are collectively called *encoded character types*.

2 Functions with template parameters named `EcharT` shall not participate in overload resolution unless `EcharT` is one of the encoded character types.

3 Template parameters named `InputIterator` shall meet the input iterator requirements (24.2.3) and shall have a value type that is one of the encoded character types.

4 [*Note:* Use of an encoded character type implies an associated encoding. Since `signed char` and `unsigned char` have no implied encoding, they are not included as permitted types. — *end note*]

5 Template parameters named `Allocator` shall meet the Allocator requirements (17.5.3.5).

27.10.5.1 Namespaces and headers [fs.req.namespace]

1 Unless otherwise specified, references to entities described in this subclause are assumed to be qualified with `::std::filesystem::`.

27.10.6 Header `<filesystem>` synopsis [fs.filesystem.syn]

```
namespace std::filesystem {
  // 27.10.8, paths
  class path;

  // 27.10.8.6, path non-member functions
  void swap(path& lhs, path& rhs) noexcept;
  size_t hash_value(const path& p) noexcept;

  bool operator==(const path& lhs, const path& rhs) noexcept;
  bool operator!=(const path& lhs, const path& rhs) noexcept;
  bool operator< (const path& lhs, const path& rhs) noexcept;
  bool operator<=(const path& lhs, const path& rhs) noexcept;
  bool operator> (const path& lhs, const path& rhs) noexcept;
  bool operator>=(const path& lhs, const path& rhs) noexcept;

  path operator/ (const path& lhs, const path& rhs);

  // 27.10.8.6.1, path inserter and extractor
  template <class charT, class traits>
    basic_ostream<charT, traits>&
      operator<<(basic_ostream<charT, traits>& os, const path& p);
  template <class charT, class traits>
```

§ 27.10.6 1228

```
  basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, path& p);

// 27.10.8.6.2, path factory functions
template <class Source>
  path u8path(const Source& source);
template <class InputIterator>
  path u8path(InputIterator first, InputIterator last);

// 27.10.9, filesystem errors
class filesystem_error;

// 27.10.12, directory entries
class directory_entry;

// 27.10.13, directory iterators
class directory_iterator;

// 27.10.13.2, range access for directory iterators
directory_iterator begin(directory_iterator iter) noexcept;
directory_iterator end(const directory_iterator&) noexcept;

// 27.10.14, recursive directory iterators
class recursive_directory_iterator;

// 27.10.14.2, range access for recursive directory iterators
recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;
recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;

// 27.10.11, file status
class file_status;

struct space_info {
  uintmax_t capacity;
  uintmax_t free;
  uintmax_t available;
};

// 27.10.10, enumerations
enum class file_type;
enum class perms;
enum class copy_options;
enum class directory_options;

using file_time_type = chrono::time_point<trivial-clock>;

// 27.10.15, filesystem operations
path absolute(const path& p, const path& base = current_path());

path canonical(const path& p, const path& base = current_path());
path canonical(const path& p, error_code& ec);
path canonical(const path& p, const path& base, error_code& ec);

void copy(const path& from, const path& to);
void copy(const path& from, const path& to, error_code& ec) noexcept;
```



```
  basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, path& p);

// 27.10.8.6.2, path factory functions
template <class Source>
  path u8path(const Source& source);
template <class InputIterator>
  path u8path(InputIterator first, InputIterator last);

// 27.10.9, filesystem errors
class filesystem_error;

// 27.10.12, directory entries
class directory_entry;

// 27.10.13, directory iterators
class directory_iterator;

// 27.10.13.2, range access for directory iterators
directory_iterator begin(directory_iterator iter) noexcept;
directory_iterator end(const directory_iterator&) noexcept;

// 27.10.14, recursive directory iterators
class recursive_directory_iterator;

// 27.10.14.2, range access for recursive directory iterators
recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;
recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;

// 27.10.11, file status
class file_status;

struct space_info {
  uintmax_t capacity;
  uintmax_t free;
  uintmax_t available;
};

// 27.10.10, enumerations
enum class file_type;
enum class perms;
enum class copy_options;
enum class directory_options;

using file_time_type = chrono::time_point<trivial-clock>;

// 27.10.15, filesystem operations
path absolute(const path& p, const path& base = current_path());

path canonical(const path& p, const path& base = current_path());
path canonical(const path& p, error_code& ec);
path canonical(const path& p, const path& base, error_code& ec);

void copy(const path& from, const path& to);
void copy(const path& from, const path& to, error_code& ec) noexcept;
```

Table 129 — Regular expression traits class requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition
`u.imbue(loc)`	`X::locale_type`	Imbues `u` with the locale `loc` and returns the previous locale used by `u` if any.
`v.getloc()`	`X::locale_type`	Returns the current locale used by `v`, if any.

5 [*Note:* Class template `regex_traits` satisfies the requirements for a regular expression traits class when it is specialized for `char` or `wchar_t`. This class template is described in the header `<regex>`, and is described in Clause 28.7. — *end note*]

28.4 Header <regex> synopsis [re.syn]

```
#include <initializer_list>

namespace std {
  // 28.5, regex constants
  namespace regex_constants {
    using syntax_option_type = T1;
    using match_flag_type = T2;
    using error_type = T3;
  }

  // 28.6, class regex_error
  class regex_error;

  // 28.7, class template regex_traits
  template <class charT> struct regex_traits;

  // 28.8, class template basic_regex
  template <class charT, class traits = regex_traits<charT>> class basic_regex;

  using regex  = basic_regex<char>;
  using wregex = basic_regex<wchar_t>;

  // 28.8.6, basic_regex swap
  template <class charT, class traits>
    void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);

  // 28.9, class template sub_match
  template <class BidirectionalIterator>
    class sub_match;

  using csub_match  = sub_match<const char*>;
  using wcsub_match = sub_match<const wchar_t*>;
  using ssub_match  = sub_match<string::const_iterator>;
  using wssub_match = sub_match<wstring::const_iterator>;

  // 28.9.2, sub_match non-member operators
  template <class BiIter>
    bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
  template <class BiIter>
    bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```


©ISO/IEC Dxxxx

Table 129 — Regular expression traits class requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition
`u.imbue(loc)`	`X::locale_type`	Imbues `u` with the locale `loc` and returns the previous locale used by `u` if any.
`v.getloc()`	`X::locale_type`	Returns the current locale used by `v`, if any.

5 [*Note:* Class template `regex_traits` satisfies the requirements for a regular expression traits class when it is specialized for `char` or `wchar_t`. This class template is described in the header `<regex>`, and is described in Clause 28.7. — *end note*]

28.4 Header <regex> synopsis [re.syn]

```
#include <initializer_list>

namespace std {
  // 28.5, regex constants
  namespace regex_constants {
    using syntax_option_type = T1;
    using match_flag_type = T2;
    using error_type = T3;
  }

  // 28.6, class regex_error
  class regex_error;

  // 28.7, class template regex_traits
  template <class charT> struct regex_traits;

  // 28.8, class template basic_regex
  template <class charT, class traits = regex_traits<charT>> class basic_regex;

  using regex  = basic_regex<char>;
  using wregex = basic_regex<wchar_t>;

  // 28.8.6, basic_regex swap
  template <class charT, class traits>
    void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);

  // 28.9, class template sub_match
  template <class BidirectionalIterator>
    class sub_match;

  using csub_match  = sub_match<const char*>;
  using wcsub_match = sub_match<const wchar_t*>;
  using ssub_match  = sub_match<string::const_iterator>;
  using wssub_match = sub_match<wstring::const_iterator>;

  // 28.9.2, sub_match non-member operators
  template <class BiIter>
    bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
  template <class BiIter>
    bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

```
  bool operator>=(const typename iterator_traits<BiIter>::value_type& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(const typename iterator_traits<BiIter>::value_type& lhs,
                  const sub_match<BiIter>& rhs);

template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  const typename iterator_traits<BiIter>::value_type& rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  const typename iterator_traits<BiIter>::value_type& rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 const typename iterator_traits<BiIter>::value_type& rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 const typename iterator_traits<BiIter>::value_type& rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  const typename iterator_traits<BiIter>::value_type& rhs);
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  const typename iterator_traits<BiIter>::value_type& rhs);

template <class charT, class ST, class BiIter>
  basic_ostream<charT, ST>&
    operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

// 28.10, class template match_results
template <class BidirectionalIterator,
          class Allocator = allocator<sub_match<BidirectionalIterator>>>
  class match_results;

using cmatch  = match_results<const char*>;
using wcmatch = match_results<const wchar_t*>;
using smatch  = match_results<string::const_iterator>;
using wsmatch = match_results<wstring::const_iterator>;

// match_results comparisons
template <class BidirectionalIterator, class Allocator>
  bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);
template <class BidirectionalIterator, class Allocator>
  bool operator!=(const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);

// 28.10.7, match_results swap
template <class BidirectionalIterator, class Allocator>
  void swap(match_results<BidirectionalIterator, Allocator>& m1,
            match_results<BidirectionalIterator, Allocator>& m2);

// 28.11.2, function template regex_match
template <class BidirectionalIterator, class Allocator, class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
```



```
  bool operator>=(const typename iterator_traits<BiIter>::value_type& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(const typename iterator_traits<BiIter>::value_type& lhs,
                  const sub_match<BiIter>& rhs);

template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  const typename iterator_traits<BiIter>::value_type& rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  const typename iterator_traits<BiIter>::value_type& rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 const typename iterator_traits<BiIter>::value_type& rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 const typename iterator_traits<BiIter>::value_type& rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  const typename iterator_traits<BiIter>::value_type& rhs);
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  const typename iterator_traits<BiIter>::value_type& rhs);

template <class charT, class ST, class BiIter>
  basic_ostream<charT, ST>&
    operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

// 28.10, class template match_results
template <class BidirectionalIterator,
          class Allocator = allocator<sub_match<BidirectionalIterator>>>
  class match_results;

using cmatch  = match_results<const char*>;
using wcmatch = match_results<const wchar_t*>;
using smatch  = match_results<string::const_iterator>;
using wsmatch = match_results<wstring::const_iterator>;

// match_results comparisons
template <class BidirectionalIterator, class Allocator>
  bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);
template <class BidirectionalIterator, class Allocator>
  bool operator!=(const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);

// 28.10.7, match_results swap
template <class BidirectionalIterator, class Allocator>
  void swap(match_results<BidirectionalIterator, Allocator>& m1,
            match_results<BidirectionalIterator, Allocator>& m2);

// 28.11.2, function template regex_match
template <class BidirectionalIterator, class Allocator, class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
```

```
                  match_results<BidirectionalIterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class BidirectionalIterator, class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class charT, class Allocator, class traits>
  bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                  match_results<typename basic_string<charT, ST, SA>::const_iterator,
                                Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>&&,
                  match_results<typename basic_string<charT, ST, SA>::const_iterator,
                                Allocator>&,
                  const basic_regex<charT, traits>&,
                  regex_constants::match_flag_type = regex_constants::match_default) = delete;
template <class charT, class traits>
  bool regex_match(const charT* str,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

// 28.11.3, function template regex_search
template <class BidirectionalIterator, class Allocator, class charT, class traits>
  bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                   match_results<BidirectionalIterator, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template <class BidirectionalIterator, class charT, class traits>
  bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template <class charT, class Allocator, class traits>
  bool regex_search(const charT* str,
                   match_results<const charT*, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template <class charT, class traits>
  bool regex_search(const charT* str,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
```



```
                  match_results<BidirectionalIterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class BidirectionalIterator, class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class charT, class Allocator, class traits>
  bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                  match_results<typename basic_string<charT, ST, SA>::const_iterator,
                                Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>&&,
                  match_results<typename basic_string<charT, ST, SA>::const_iterator,
                                Allocator>&,
                  const basic_regex<charT, traits>&,
                  regex_constants::match_flag_type = regex_constants::match_default) = delete;
template <class charT, class traits>
  bool regex_match(const charT* str,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

// 28.11.3, function template regex_search
template <class BidirectionalIterator, class Allocator, class charT, class traits>
  bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                   match_results<BidirectionalIterator, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template <class BidirectionalIterator, class charT, class traits>
  bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template <class charT, class Allocator, class traits>
  bool regex_search(const charT* str,
                   match_results<const charT*, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template <class charT, class traits>
  bool regex_search(const charT* str,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
```

```
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    match_results<typename basic_string<charT, ST, SA>::const_iterator,
                                  Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>&&,
                    match_results<typename basic_string<charT, ST, SA>::const_iterator,
                                  Allocator>&,
                    const basic_regex<charT, traits>&,
                    regex_constants::match_flag_type
                      = regex_constants::match_default) = delete;

// 28.11.4, function template regex_replace
template <class OutputIterator, class BidirectionalIterator,
          class traits, class charT, class ST, class SA>
  OutputIterator
    regex_replace(OutputIterator out,
                  BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  const basic_string<charT, ST, SA>& fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class OutputIterator, class BidirectionalIterator, class traits, class charT>
  OutputIterator
    regex_replace(OutputIterator out,
                  BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  const charT* fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT, class ST, class SA, class FST, class FSA>
  basic_string<charT, ST, SA>
    regex_replace(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  const basic_string<charT, FST, FSA>& fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT, class ST, class SA>
  basic_string<charT, ST, SA>
    regex_replace(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  const charT* fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT, class ST, class SA>
  basic_string<charT>
    regex_replace(const charT* s,
                  const basic_regex<charT, traits>& e,
                  const basic_string<charT, ST, SA>& fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT>
  basic_string<charT>
    regex_replace(const charT* s,
                  const basic_regex<charT, traits>& e,
                  const charT* fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
```



```
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    match_results<typename basic_string<charT, ST, SA>::const_iterator,
                                  Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags = regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>&&,
                    match_results<typename basic_string<charT, ST, SA>::const_iterator,
                                  Allocator>&,
                    const basic_regex<charT, traits>&,
                    regex_constants::match_flag_type
                      = regex_constants::match_default) = delete;

// 28.11.4, function template regex_replace
template <class OutputIterator, class BidirectionalIterator,
          class traits, class charT, class ST, class SA>
  OutputIterator
    regex_replace(OutputIterator out,
                  BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  const basic_string<charT, ST, SA>& fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class OutputIterator, class BidirectionalIterator, class traits, class charT>
  OutputIterator
    regex_replace(OutputIterator out,
                  BidirectionalIterator first, BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  const charT* fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT, class ST, class SA, class FST, class FSA>
  basic_string<charT, ST, SA>
    regex_replace(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  const basic_string<charT, FST, FSA>& fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT, class ST, class SA>
  basic_string<charT, ST, SA>
    regex_replace(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  const charT* fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT, class ST, class SA>
  basic_string<charT>
    regex_replace(const charT* s,
                  const basic_regex<charT, traits>& e,
                  const basic_string<charT, ST, SA>& fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
template <class traits, class charT>
  basic_string<charT>
    regex_replace(const charT* s,
                  const basic_regex<charT, traits>& e,
                  const charT* fmt,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
```

```
  // 28.12.1, class template regex_iterator
  template <class BidirectionalIterator,
            class charT = typename iterator_traits<BidirectionalIterator>::value_type,
            class traits = regex_traits<charT>>
    class regex_iterator;

  using cregex_iterator  = regex_iterator<const char*>;
  using wcregex_iterator = regex_iterator<const wchar_t*>;
  using sregex_iterator  = regex_iterator<string::const_iterator>;
  using wsregex_iterator = regex_iterator<wstring::const_iterator>;

  // 28.12.2, class template regex_token_iterator
  template <class BidirectionalIterator,
            class charT = typename iterator_traits<BidirectionalIterator>::value_type,
            class traits = regex_traits<charT>>
    class regex_token_iterator;

  using cregex_token_iterator  = regex_token_iterator<const char*>;
  using wcregex_token_iterator = regex_token_iterator<const wchar_t*>;
  using sregex_token_iterator  = regex_token_iterator<string::const_iterator>;
  using wsregex_token_iterator = regex_token_iterator<wstring::const_iterator>;

  namespace pmr {
    template <class BidirectionalIterator>
      using match_results =
        std::match_results<BidirectionalIterator,
                           polymorphic_allocator<sub_match<BidirectionalIterator>>>;

    using cmatch  = match_results<const char*>;
    using wcmatch = match_results<const wchar_t*>;
    using smatch  = match_results<string::const_iterator>;
    using wsmatch = match_results<wstring::const_iterator>;
  }
}
```

28.5 Namespace `std::regex_constants` [re.const]

1 The namespace `std::regex_constants` holds symbolic constants used by the regular expression library. This namespace provides three types, `syntax_option_type`, `match_flag_type`, and `error_type`, along with several constants of these types.

28.5.1 Bitmask type `syntax_option_type` [re.synopt]

```
namespace std::regex_constants {
  using syntax_option_type = T1;
  constexpr syntax_option_type icase = unspecified;
  constexpr syntax_option_type nosubs = unspecified;
  constexpr syntax_option_type optimize = unspecified;
  constexpr syntax_option_type collate = unspecified;
  constexpr syntax_option_type ECMAScript = unspecified;
  constexpr syntax_option_type basic = unspecified;
  constexpr syntax_option_type extended = unspecified;
  constexpr syntax_option_type awk = unspecified;
  constexpr syntax_option_type grep = unspecified;
  constexpr syntax_option_type egrep = unspecified;
  constexpr syntax_option_type multiline = unspecified;
}
```



```
  // 28.12.1, class template regex_iterator
  template <class BidirectionalIterator,
            class charT = typename iterator_traits<BidirectionalIterator>::value_type,
            class traits = regex_traits<charT>>
    class regex_iterator;

  using cregex_iterator  = regex_iterator<const char*>;
  using wcregex_iterator = regex_iterator<const wchar_t*>;
  using sregex_iterator  = regex_iterator<string::const_iterator>;
  using wsregex_iterator = regex_iterator<wstring::const_iterator>;

  // 28.12.2, class template regex_token_iterator
  template <class BidirectionalIterator,
            class charT = typename iterator_traits<BidirectionalIterator>::value_type,
            class traits = regex_traits<charT>>
    class regex_token_iterator;

  using cregex_token_iterator  = regex_token_iterator<const char*>;
  using wcregex_token_iterator = regex_token_iterator<const wchar_t*>;
  using sregex_token_iterator  = regex_token_iterator<string::const_iterator>;
  using wsregex_token_iterator = regex_token_iterator<wstring::const_iterator>;

  namespace pmr {
    template <class BidirectionalIterator>
      using match_results =
        std::match_results<BidirectionalIterator,
                           polymorphic_allocator<sub_match<BidirectionalIterator>>>;

    using cmatch  = match_results<const char*>;
    using wcmatch = match_results<const wchar_t*>;
    using smatch  = match_results<string::const_iterator>;
    using wsmatch = match_results<wstring::const_iterator>;
  }
}
```

28.5 Namespace `std::regex_constants` [re.const]

1 The namespace `std::regex_constants` holds symbolic constants used by the regular expression library. This namespace provides three types, `syntax_option_type`, `match_flag_type`, and `error_type`, along with several constants of these types.

28.5.1 Bitmask type `syntax_option_type` [re.synopt]

```
namespace std::regex_constants {
  using syntax_option_type = T1;
  constexpr syntax_option_type icase = unspecified;
  constexpr syntax_option_type nosubs = unspecified;
  constexpr syntax_option_type optimize = unspecified;
  constexpr syntax_option_type collate = unspecified;
  constexpr syntax_option_type ECMAScript = unspecified;
  constexpr syntax_option_type basic = unspecified;
  constexpr syntax_option_type extended = unspecified;
  constexpr syntax_option_type awk = unspecified;
  constexpr syntax_option_type grep = unspecified;
  constexpr syntax_option_type egrep = unspecified;
  constexpr syntax_option_type multiline = unspecified;
}
```

```
  constexpr match_flag_type format_sed = unspecified;
  constexpr match_flag_type format_no_copy = unspecified;
  constexpr match_flag_type format_first_only = unspecified;
}
```

1 The type `match_flag_type` is an implementation-defined bitmask type (17.4.2.1.4). The constants of that type, except for `match_default` and `format_default`, are bitmask elements. The `match_default` and `format_default` constants are empty bitmasks. Matching a regular expression against a sequence of characters `[first, last)` proceeds according to the rules of the grammar specified for the regular expression object, modified according to the effects listed in Table 131 for any bitmask elements set.

Table 131 — `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence `[first, last)`.

Element	Effect(s) if set
`match_not_bol`	The first character in the sequence `[first, last)` shall be treated as though it is not at the beginning of a line, so the character `^` in the regular expression shall not match `[first, first)`.
`match_not_eol`	The last character in the sequence `[first, last)` shall be treated as though it is not at the end of a line, so the character `"$"` in the regular expression shall not match `[last, last)`.
`match_not_bow`	The expression `"\\b"` shall not match the sub-sequence `[first, first)`.
`match_not_eow`	The expression `"\\b"` shall not match the sub-sequence `[last, last)`.
`match_any`	If more than one match is possible then any match is an acceptable result.
`match_not_null`	The expression shall not match an empty sequence.
`match_continuous`	The expression shall only match a sub-sequence that begins at `first`.
`match_prev_avail`	`--first` is a valid iterator position. When this flag is set the flags match_not_bol and match_not_bow shall be ignored by the regular expression algorithms 28.11 and iterators 28.12.
`format_default`	When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the ECMAScript replace function in ECMA-262, part 15.5.4.11 String.prototype.replace. In addition, during search and replace operations all non-overlapping occurrences of the regular expression shall be located and replaced, and sections of the input that did not match the expression shall be copied unchanged to the output string.
`format_sed`	When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the sed utility in POSIX.
`format_no_copy`	During a search and replace operation, sections of the character container sequence being searched that do not match the regular expression shall not be copied to the output string.
`format_first_only`	When specified during a search and replace operation, only the first occurrence of the regular expression shall be replaced.

28.5.3 Implementation-defined `error_type` [re.err]

```
namespace std::regex_constants {
  using error_type = T3;
  constexpr error_type error_collate = unspecified;
  constexpr error_type error_ctype = unspecified;
  constexpr error_type error_escape = unspecified;
```


©ISO/IEC Dxxxx

```
  constexpr match_flag_type format_sed = unspecified;
  constexpr match_flag_type format_no_copy = unspecified;
  constexpr match_flag_type format_first_only = unspecified;
}
```

1 The type `match_flag_type` is an implementation-defined bitmask type (17.4.2.1.4). The constants of that type, except for `match_default` and `format_default`, are bitmask elements. The `match_default` and `format_default` constants are empty bitmasks. Matching a regular expression against a sequence of characters `[first, last)` proceeds according to the rules of the grammar specified for the regular expression object, modified according to the effects listed in Table 131 for any bitmask elements set.

Table 131 — `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence `[first, last)`.

Element	Effect(s) if set
`match_not_bol`	The first character in the sequence `[first, last)` shall be treated as though it is not at the beginning of a line, so the character `^` in the regular expression shall not match `[first, first)`.
`match_not_eol`	The last character in the sequence `[first, last)` shall be treated as though it is not at the end of a line, so the character `"$"` in the regular expression shall not match `[last, last)`.
`match_not_bow`	The expression `"\\b"` shall not match the sub-sequence `[first, first)`.
`match_not_eow`	The expression `"\\b"` shall not match the sub-sequence `[last, last)`.
`match_any`	If more than one match is possible then any match is an acceptable result.
`match_not_null`	The expression shall not match an empty sequence.
`match_continuous`	The expression shall only match a sub-sequence that begins at `first`.
`match_prev_avail`	`--first` is a valid iterator position. When this flag is set the flags `match_not_bol` and `match_not_bow` shall be ignored by the regular expression algorithms 28.11 and iterators 28.12.
`format_default`	When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the ECMAScript replace function in ECMA-262, part 15.5.4.11 String.prototype.replace. In addition, during search and replace operations all non-overlapping occurrences of the regular expression shall be located and replaced, and sections of the input that did not match the expression shall be copied unchanged to the output string.
`format_sed`	When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the sed utility in POSIX.
`format_no_copy`	During a search and replace operation, sections of the character container sequence being searched that do not match the regular expression shall not be copied to the output string.
`format_first_only`	When specified during a search and replace operation, only the first occurrence of the regular expression shall be replaced.

28.5.3 Implementation-defined `error_type` [re.err]

```
namespace std::regex_constants {
  using error_type = T3;
  constexpr error_type error_collate = unspecified;
  constexpr error_type error_ctype = unspecified;
  constexpr error_type error_escape = unspecified;
```

§ 28.5.3 1289

nested or local functions.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:

```
struct Y;                       // struct Y and struct X are at the same scope
struct X {
  struct Y { /* ... */ } y;
};
```

All the definitions of C struct types enclosed in other struct definitions and accessed outside the scope of the enclosing struct could be exported to the scope of the enclosing struct. Note: this is a consequence of the difference in scope rules, which is documented in 3.3.
How widely used: Seldom.

9.2.6
Change: In C++, a typedef name may not be redeclared in a class definition after being used in that definition.

Example:

```
typedef int I;
struct S {
  I i;
  int I;                    // valid C, invalid C++
};
```

Rationale: When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Either the type or the struct member has to be renamed.
How widely used: Seldom.

C.1.9 Clause 12: special member functions [diff.special]

12.8
Change: Copying volatile objects.

The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:

```
struct X { int i; };
volatile struct X x1 = {0};
struct X x2 = x1;               // invalid C++
struct X x3;
x3 = x1;                        // also invalid C++
```

Rationale: Several alternatives were debated at length. Changing the parameter to `volatile const X&` would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. If volatile semantics are required for the copy, a

©ISO/IEC Dxxxx

nested or local functions.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:

```
struct Y;                       // struct Y and struct X are at the same scope
struct X {
  struct Y { /* ... */ } y;
};
```

All the definitions of C struct types enclosed in other struct definitions and accessed outside the scope of the enclosing struct could be exported to the scope of the enclosing struct. Note: this is a consequence of the difference in scope rules, which is documented in 3.3.
How widely used: Seldom.

9.2.6
Change: In C++, a typedef name may not be redeclared in a class definition after being used in that definition.

Example:

```
typedef int I;
struct S {
  I i;
  int I;                    // valid C, invalid C++
};
```

Rationale: When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Either the type or the struct member has to be renamed.
How widely used: Seldom.

C.1.9 Clause 12: special member functions [diff.special]

12.8
Change: Copying volatile objects.

The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:

```
struct X { int i; };
volatile struct X x1 = {0};
struct X x2 = x1;               // invalid C++
struct X x3;
x3 = x1;                        // also invalid C++
```

Rationale: Several alternatives were debated at length. Changing the parameter to `volatile const X&` would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. If volatile semantics are required for the copy, a

§ C.1.9 1424