multibyte strings (NTMBS s) (17.5.2.1.4.2) and `argv[0]` shall be the pointer to the initial character of a NTMBS that represents the name used to invoke the program or `""`. The value of `argc` shall be non-negative. The value of `argv[argc]` shall be 0. [*Note:* It is recommended that any further (optional) parameters be added after `argv`. — *end note*]

3 The function `main` shall not be used within a program. The linkage (3.5) of `main` is implementation-defined. A program that defines `main` as deleted or that declares `main` to be `inline`, `static`, or `constexpr` is ill-formed. The `main` function shall not be declared with a *linkage-specification* (7.5). A program that declares a variable `main` at global scope or that declares the name `main` with C language linkage (in any namespace) is ill-formed. The name `main` is not otherwise reserved. [*Example:* member functions, classes, and enumerations can be called `main`, as can entities in other namespaces. — *end example*]

4 Terminating the program without leaving the current block (e.g., by calling the function `std::exit(int)` (18.5)) does not destroy any objects with automatic storage duration (12.4). If `std::exit` is called to end a program during the destruction of an object with static or thread storage duration, the program has undefined behavior.

5 A return statement in `main` has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling `std::exit` with the return value as the argument. If control flows off the end of the *compound-statement* of `main`, the effect is equivalent to a `return` with operand `0` (see also 15.3).

3.6.2 Static initialization [basic.start.static]

1 Variables with static storage duration are initialized as a consequence of program initiation. Variables with thread storage duration are initialized as a consequence of thread execution. Within each of these phases of initiation, initialization occurs as follows.

2 A *constant initializer* for an object `o` is an expression that is a constant expression, except that it may also invoke `constexpr` constructors for `o` and its subobjects even if those objects are of non-literal class types. [*Note:* Such a class may have a non-trivial destructor — *end note*] *Constant initialization* is performed:

(2.1) — if each full-expression (including implicit conversions) that appears in the initializer of a reference with static or thread storage duration is a constant expression (5.20) and the reference is bound to a glvalue designating an object with static storage duration, to a temporary object (see 12.2) or subobject thereof, or to a function;

(2.2) — if an object with static or thread storage duration is initialized by a constructor call, and if the initialization full-expression is a constant initializer for the object;

(2.3) — if an object with static or thread storage duration is not initialized by a constructor call and if either the object is value-initialized or every full-expression that appears in its initializer is a constant expression.

If constant initialization is not performed, a variable with static storage duration (3.7.1) or thread storage duration (3.7.2) is zero-initialized (8.6). Together, zero-initialization and constant initialization are called *static initialization*; all other initialization is *dynamic initialization*. Static initialization shall be performed before any dynamic initialization takes place. [*Note:* The dynamic initialization of non-local variables is described in 3.6.3; that of local static variables is described in 6.7. — *end note*]

3 An implementation is permitted to perform the initialization of a variable with static or thread storage duration as a static initialization even if such initialization is not required to be done statically, provided that

(3.1) — the dynamic version of the initialization does not change the value of any other object of static or thread storage duration prior to its initialization, and

(3.2) — the static version of the initialization produces the same value in the initialized variable as would be produced by the dynamic initialization if all variables not required to be initialized statically were initialized dynamically.

multibyte strings (NTMBS s) (17.5.2.1.4.2) and `argv[0]` shall be the pointer to the initial character of a NTMBS that represents the name used to invoke the program or `""`. The value of `argc` shall be non-negative. The value of `argv[argc]` shall be `nullptr`. [*Note:* It is recommended that any further (optional) parameters be added after `argv`. — *end note*]

3 The function `main` shall not be used within a program. The linkage (3.5) of `main` is implementation-defined. A program that defines `main` as deleted or that declares `main` to be `inline`, `static`, or `constexpr` is ill-formed. The `main` function shall not be declared with a *linkage-specification* (7.5). A program that declares a variable `main` at global scope or that declares the name `main` with C language linkage (in any namespace) is ill-formed. The name `main` is not otherwise reserved. [*Example:* member functions, classes, and enumerations can be called `main`, as can entities in other namespaces. — *end example*]

4 Terminating the program without leaving the current block (e.g., by calling the function `std::exit(int)` (18.5)) does not destroy any objects with automatic storage duration (12.4). If `std::exit` is called to end a program during the destruction of an object with static or thread storage duration, the program has undefined behavior.

5 A return statement in `main` has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling `std::exit` with the return value as the argument. If control flows off the end of the *compound-statement* of `main`, the effect is equivalent to a `return` with operand `0` (see also 15.3).

3.6.2 Static initialization [basic.start.static]

1 Variables with static storage duration are initialized as a consequence of program initiation. Variables with thread storage duration are initialized as a consequence of thread execution. Within each of these phases of initiation, initialization occurs as follows.

2 A *constant initializer* for an object `o` is an expression that is a constant expression, except that it may also invoke `constexpr` constructors for `o` and its subobjects even if those objects are of non-literal class types. [*Note:* Such a class may have a non-trivial destructor — *end note*] *Constant initialization* is performed:

(2.1) — if each full-expression (including implicit conversions) that appears in the initializer of a reference with static or thread storage duration is a constant expression (5.20) and the reference is bound to a glvalue designating an object with static storage duration, to a temporary object (see 12.2) or subobject thereof, or to a function;

(2.2) — if an object with static or thread storage duration is initialized by a constructor call, and if the initialization full-expression is a constant initializer for the object;

(2.3) — if an object with static or thread storage duration is not initialized by a constructor call and if either the object is value-initialized or every full-expression that appears in its initializer is a constant expression.

If constant initialization is not performed, a variable with static storage duration (3.7.1) or thread storage duration (3.7.2) is zero-initialized (8.6). Together, zero-initialization and constant initialization are called *static initialization*; all other initialization is *dynamic initialization*. Static initialization shall be performed before any dynamic initialization takes place. [*Note:* The dynamic initialization of non-local variables is described in 3.6.3; that of local static variables is described in 6.7. — *end note*]

3 An implementation is permitted to perform the initialization of a variable with static or thread storage duration as a static initialization even if such initialization is not required to be done statically, provided that

(3.1) — the dynamic version of the initialization does not change the value of any other object of static or thread storage duration prior to its initialization, and

(3.2) — the static version of the initialization produces the same value in the initialized variable as would be produced by the dynamic initialization if all variables not required to be initialized statically were initialized dynamically.

8 If C is the class type to which T points or refers, the run-time check logically executes as follows:

(8.1) — If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class subobject of a C object, and if only one object of type C is derived from the subobject pointed (referred) to by v the result points (refers) to that C object.

(8.2) — Otherwise, if v points (refers) to a public base class subobject of the most derived object, and the type of the most derived object has a base class, of type C, that is unambiguous and public, the result points (refers) to the C subobject of the most derived object.

(8.3) — Otherwise, the run-time check *fails*.

9 The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws an exception (15.1) of a type that would match a handler (15.3) of type std::bad_cast (18.7.3).

[*Example:*

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D  d;
  B* bp = (B*)&d;                 // cast needed to break protection
  A* ap = &d;                     // public derivation, no cast needed
  D& dr = dynamic_cast<D&>(*bp);  // fails
  ap = dynamic_cast<A*>(bp);      // fails
  bp = dynamic_cast<B*>(ap);      // fails
  ap = dynamic_cast<A*>(&d);      // succeeds
  bp = dynamic_cast<B*>(&d);      // ill-formed (not a run-time check)
}

class E : public D, public B { };
class F : public E, public D { };
void h() {
  F  f;
  A* ap = &f;                     // succeeds: finds unique A
  D* dp = dynamic_cast<D*>(ap);   // fails: yields 0
                                  // f has two D subobjects
  E* ep = (E*)ap;                 // ill-formed: cast from virtual base
  E* ep1 = dynamic_cast<E*>(ap);  // succeeds
}
```

— *end example*] [*Note:* 12.7 describes the behavior of a dynamic_cast applied to an object under construction or destruction. — *end note*]

5.2.8 Type identification [expr.typeid]

1 The result of a typeid expression is an lvalue of static type const std::type_info (18.7.2) and dynamic type const std::type_info or const *name* where *name* is an implementation-defined class publicly derived from std::type_info which preserves the behavior described in 18.7.2.[69] The lifetime of the object referred to by the lvalue extends to the end of the program. Whether or not the destructor is called for the std::type_info object at the end of the program is unspecified.

2 When typeid is applied to a glvalue expression whose type is a polymorphic class type (10.3), the result refers to a std::type_info object representing the type of the most derived object (1.8) (that is, the dynamic

69) The recommended name for such a class is extended_type_info.

8 If C is the class type to which T points or refers, the run-time check logically executes as follows:

(8.1) — If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class subobject of a C object, and if only one object of type C is derived from the subobject pointed (referred) to by v the result points (refers) to that C object.

(8.2) — Otherwise, if v points (refers) to a public base class subobject of the most derived object, and the type of the most derived object has a base class, of type C, that is unambiguous and public, the result points (refers) to the C subobject of the most derived object.

(8.3) — Otherwise, the run-time check *fails*.

9 The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws an exception (15.1) of a type that would match a handler (15.3) of type std::bad_cast (18.7.3).

[*Example:*

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D  d;
  B* bp = (B*)&d;                 // cast needed to break protection
  A* ap = &d;                     // public derivation, no cast needed
  D& dr = dynamic_cast<D&>(*bp);  // fails
  ap = dynamic_cast<A*>(bp);      // fails
  bp = dynamic_cast<B*>(ap);      // fails
  ap = dynamic_cast<A*>(&d);      // succeeds
  bp = dynamic_cast<B*>(&d);      // ill-formed (not a run-time check)
}

class E : public D, public B { };
class F : public E, public D { };
void h() {
  F  f;
  A* ap = &f;                     // succeeds: finds unique A
  D* dp = dynamic_cast<D*>(ap);   // fails: yields nullptr
                                  // f has two D subobjects
  E* ep = (E*)ap;                 // ill-formed: cast from virtual base
  E* ep1 = dynamic_cast<E*>(ap);  // succeeds
}
```

— *end example*] [*Note:* 12.7 describes the behavior of a dynamic_cast applied to an object under construction or destruction. — *end note*]

5.2.8 Type identification [expr.typeid]

1 The result of a typeid expression is an lvalue of static type const std::type_info (18.7.2) and dynamic type const std::type_info or const *name* where *name* is an implementation-defined class publicly derived from std::type_info which preserves the behavior described in 18.7.2.[69] The lifetime of the object referred to by the lvalue extends to the end of the program. Whether or not the destructor is called for the std::type_info object at the end of the program is unspecified.

2 When typeid is applied to a glvalue expression whose type is a polymorphic class type (10.3), the result refers to a std::type_info object representing the type of the most derived object (1.8) (that is, the dynamic

69) The recommended name for such a class is extended_type_info.

— *end example*]

3 An lvalue of type "*cv1* `T1`" can be cast to type "rvalue reference to *cv2* `T2`" if "*cv2* `T2`" is reference-compatible with "*cv1* `T1`" (8.6.3). If the value is not a bit-field, the result refers to the object or the specified base class subobject thereof; otherwise, the lvalue-to-rvalue conversion (4.1) is applied to the bit-field and the resulting prvalue is used as the *expression* of the `static_cast` for the remainder of this section. If `T2` is an inaccessible (Clause 11) or ambiguous (10.2) base class of `T1`, a program that necessitates such a cast is ill-formed.

4 An expression `e` can be explicitly converted to a type `T` if there is an implicit conversion sequence (13.3.3.1) from `e` to `T`, or if overload resolution for a direct-initialization (8.6) of an object or reference of type `T` from `e` would find at least one viable function (13.3.2). If `T` is a reference type, the effect is the same as performing the declaration and initialization

```
T t(e);
```

for some invented temporary variable `t` (8.6) and then using the temporary variable as the result of the conversion. Otherwise, the result object is direct-initialized from `e`. [*Note:* The conversion is ill-formed when attempting to convert an expression of class type to an inaccessible or ambiguous base class. — *end note*]

5 Otherwise, the `static_cast` shall perform one of the conversions listed below. No other conversion shall be performed explicitly using a `static_cast`.

6 Any expression can be explicitly converted to type *cv* `void`, in which case it becomes a discarded-value expression (Clause 5). [*Note:* however, if the value is in a temporary object (12.2), the destructor for that object is not executed until the usual time, and the value of the object is preserved for the purpose of executing the destructor. — *end note*]

7 The inverse of any standard conversion sequence (Clause 4) not containing an lvalue-to-rvalue (4.1), array-to-pointer (4.2), function-to-pointer (4.3), null pointer (4.11), null member pointer (4.12), boolean (4.14), or function pointer (4.13) conversion, can be performed explicitly using `static_cast`. A program is ill-formed if it uses `static_cast` to perform the inverse of an ill-formed standard conversion sequence. [*Example:*

```
struct B { };
struct D : private B { };
void f() {
  static_cast<D*>((B*)0);              // Error: B is a private base of D.
  static_cast<int B::*>((int D::*)0);  // Error: B is a private base of D.
}
```

— *end example*]

8 The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) conversions are applied to the operand. Such a `static_cast` is subject to the restriction that the explicit conversion does not cast away constness (5.2.11), and the following additional rules for specific cases:

9 A value of a scoped enumeration type (7.2) can be explicitly converted to an integral type. When that type is *cv* `bool`, the resulting value is `false` if the original value is zero and `true` for all other values. For the remaining integral types, the value is unchanged if the original value can be represented by the specified type. Otherwise, the resulting value is unspecified. A value of a scoped enumeration type can also be explicitly converted to a floating-point type; the result is the same as that of converting from the original value to the floating-point type.

10 A value of integral or enumeration type can be explicitly converted to a complete enumeration type. The value is unchanged if the original value is within the range of the enumeration values (7.2). Otherwise, the behavior is undefined. A value of floating-point type can also be explicitly converted to an enumeration type. The resulting value is the same as converting the original value to the underlying type of the enumeration (4.10), and subsequently to the enumeration type.

— *end example*]

3 An lvalue of type "*cv1* `T1`" can be cast to type "rvalue reference to *cv2* `T2`" if "*cv2* `T2`" is reference-compatible with "*cv1* `T1`" (8.6.3). If the value is not a bit-field, the result refers to the object or the specified base class subobject thereof; otherwise, the lvalue-to-rvalue conversion (4.1) is applied to the bit-field and the resulting prvalue is used as the *expression* of the `static_cast` for the remainder of this section. If `T2` is an inaccessible (Clause 11) or ambiguous (10.2) base class of `T1`, a program that necessitates such a cast is ill-formed.

4 An expression `e` can be explicitly converted to a type `T` if there is an implicit conversion sequence (13.3.3.1) from `e` to `T`, or if overload resolution for a direct-initialization (8.6) of an object or reference of type `T` from `e` would find at least one viable function (13.3.2). If `T` is a reference type, the effect is the same as performing the declaration and initialization

```
T t(e);
```

for some invented temporary variable `t` (8.6) and then using the temporary variable as the result of the conversion. Otherwise, the result object is direct-initialized from `e`. [*Note:* The conversion is ill-formed when attempting to convert an expression of class type to an inaccessible or ambiguous base class. — *end note*]

5 Otherwise, the `static_cast` shall perform one of the conversions listed below. No other conversion shall be performed explicitly using a `static_cast`.

6 Any expression can be explicitly converted to type *cv* `void`, in which case it becomes a discarded-value expression (Clause 5). [*Note:* however, if the value is in a temporary object (12.2), the destructor for that object is not executed until the usual time, and the value of the object is preserved for the purpose of executing the destructor. — *end note*]

7 The inverse of any standard conversion sequence (Clause 4) not containing an lvalue-to-rvalue (4.1), array-to-pointer (4.2), function-to-pointer (4.3), null pointer (4.11), null member pointer (4.12), boolean (4.14), or function pointer (4.13) conversion, can be performed explicitly using `static_cast`. A program is ill-formed if it uses `static_cast` to perform the inverse of an ill-formed standard conversion sequence. [*Example:*

```
struct B { };
struct D : private B { };
void f() {
  static_cast<D*>((B*)nullptr);              // Error: B is a private base of D.
  static_cast<int B::*>((int D::*)nullptr);  // Error: B is a private base of D.
}
```

— *end example*]

8 The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) conversions are applied to the operand. Such a `static_cast` is subject to the restriction that the explicit conversion does not cast away constness (5.2.11), and the following additional rules for specific cases:

9 A value of a scoped enumeration type (7.2) can be explicitly converted to an integral type. When that type is *cv* `bool`, the resulting value is `false` if the original value is zero and `true` for all other values. For the remaining integral types, the value is unchanged if the original value can be represented by the specified type. Otherwise, the resulting value is unspecified. A value of a scoped enumeration type can also be explicitly converted to a floating-point type; the result is the same as that of converting from the original value to the floating-point type.

10 A value of integral or enumeration type can be explicitly converted to a complete enumeration type. The value is unchanged if the original value is within the range of the enumeration values (7.2). Otherwise, the behavior is undefined. A value of floating-point type can also be explicitly converted to an enumeration type. The resulting value is the same as converting the original value to the underlying type of the enumeration (4.10), and subsequently to the enumeration type.

11 A prvalue of type "pointer to *cv1* `B`", where `B` is a class type, can be converted to a prvalue of type "pointer to *cv2* `D`", where `D` is a class derived (Clause 10) from `B`, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. If `B` is a virtual base class of `D` or a base class of a virtual base class of `D`, or if no valid standard conversion from "pointer to `D`" to "pointer to `B`" exists (4.11), the program is ill-formed. The null pointer value (4.11) is converted to the null pointer value of the destination type. If the prvalue of type "pointer to *cv1* `B`" points to a `B` that is actually a subobject of an object of type `D`, the resulting pointer points to the enclosing object of type `D`. Otherwise, the behavior is undefined.

12 A prvalue of type "pointer to member of `D` of type *cv1* `T`" can be converted to a prvalue of type "pointer to member of `B` of type *cv2* `T`", where `B` is a base class (Clause 10) of `D`, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*.[71] If no valid standard conversion from "pointer to member of `B` of type `T`" to "pointer to member of `D` of type `T`" exists (4.12), the program is ill-formed. The null member pointer value (4.12) is converted to the null member pointer value of the destination type. If class `B` contains the original member, or is a base or derived class of the class containing the original member, the resulting pointer to member points to the original member. Otherwise, the behavior is undefined. [*Note:* although class `B` need not contain the original member, the dynamic type of the object with which indirection through the pointer to member is performed must contain the original member; see 5.5. — *end note*]

13 A prvalue of type "pointer to *cv1* `void`" can be converted to a prvalue of type "pointer to *cv2* `T`", where `T` is an object type and *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. If the original pointer value represents the address `A` of a byte in memory and `A` does not satisfy the alignment requirement of `T`, then the resulting pointer value is unspecified. Otherwise, if the original pointer value points to an object *a*, and there is an object *b* of type `T` (ignoring cv-qualification) that is pointer-interconvertible (3.9.2) with *a*, the result is a pointer to *b*. Otherwise, the pointer value is unchanged by the conversion. [*Example:*

```
T* p1 = new T;
const T* p2 = static_cast<const T*>(static_cast<void*>(p1));
bool b = p1 == p2;  // b will have the value true.
```

— *end example*]

5.2.10 Reinterpret cast [expr.reinterpret.cast]

1 The result of the expression `reinterpret_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the expression `v`. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

2 The `reinterpret_cast` operator shall not cast away constness (5.2.11). An expression of integral, enumeration, pointer, or pointer-to-member type can be explicitly converted to its own type; such a cast yields the value of its operand.

3 [*Note:* The mapping performed by `reinterpret_cast` might, or might not, produce a representation different from the original value. — *end note*]

4 A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined. [*Note:* It is intended to be unsurprising to those who know the addressing structure of the underlying machine. — *end note*] A value of type `std::nullptr_t` can be converted to an integral type; the conversion has the same meaning and validity as a conversion of `(void*)0` to the integral type. [*Note:* A `reinterpret_cast` cannot be used to convert a value of any type to the type `std::nullptr_t`. — *end note*]

71) Function types (including those used in pointer to member function types) are never cv-qualified; see 8.3.5.

11 A prvalue of type "pointer to *cv1* `B`", where `B` is a class type, can be converted to a prvalue of type "pointer to *cv2* `D`", where `D` is a class derived (Clause 10) from `B`, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. If `B` is a virtual base class of `D` or a base class of a virtual base class of `D`, or if no valid standard conversion from "pointer to `D`" to "pointer to `B`" exists (4.11), the program is ill-formed. The null pointer value (4.11) is converted to the null pointer value of the destination type. If the prvalue of type "pointer to *cv1* `B`" points to a `B` that is actually a subobject of an object of type `D`, the resulting pointer points to the enclosing object of type `D`. Otherwise, the behavior is undefined.

12 A prvalue of type "pointer to member of `D` of type *cv1* `T`" can be converted to a prvalue of type "pointer to member of `B` of type *cv2* `T`", where `B` is a base class (Clause 10) of `D`, if *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*.[71] If no valid standard conversion from "pointer to member of `B` of type `T`" to "pointer to member of `D` of type `T`" exists (4.12), the program is ill-formed. The null member pointer value (4.12) is converted to the null member pointer value of the destination type. If class `B` contains the original member, or is a base or derived class of the class containing the original member, the resulting pointer to member points to the original member. Otherwise, the behavior is undefined. [*Note:* although class `B` need not contain the original member, the dynamic type of the object with which indirection through the pointer to member is performed must contain the original member; see 5.5. — *end note*]

13 A prvalue of type "pointer to *cv1* `void`" can be converted to a prvalue of type "pointer to *cv2* `T`", where `T` is an object type and *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. If the original pointer value represents the address `A` of a byte in memory and `A` does not satisfy the alignment requirement of `T`, then the resulting pointer value is unspecified. Otherwise, if the original pointer value points to an object *a*, and there is an object *b* of type `T` (ignoring cv-qualification) that is pointer-interconvertible (3.9.2) with *a*, the result is a pointer to *b*. Otherwise, the pointer value is unchanged by the conversion. [*Example:*

```
T* p1 = new T;
const T* p2 = static_cast<const T*>(static_cast<void*>(p1));
bool b = p1 == p2;  // b will have the value true.
```

— *end example*]

5.2.10 Reinterpret cast [expr.reinterpret.cast]

1 The result of the expression `reinterpret_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the expression `v`. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

2 The `reinterpret_cast` operator shall not cast away constness (5.2.11). An expression of integral, enumeration, pointer, or pointer-to-member type can be explicitly converted to its own type; such a cast yields the value of its operand.

3 [*Note:* The mapping performed by `reinterpret_cast` might, or might not, produce a representation different from the original value. — *end note*]

4 A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined. [*Note:* It is intended to be unsurprising to those who know the addressing structure of the underlying machine. — *end note*] A value of type `std::nullptr_t` can be converted to an integral type; the conversion has the same meaning and validity as a conversion of `(void*)nullptr` to the integral type. [*Note:* A `reinterpret_cast` cannot be used to convert a value of any type to the type `std::nullptr_t`. — *end note*]

71) Function types (including those used in pointer to member function types) are never cv-qualified; see 8.3.5.

4 [*Note:* Forming a pointer to reference type is ill-formed; see 8.3.2. Forming a function pointer type is ill-formed if the function type has *cv-qualifiers* or a *ref-qualifier*; see 8.3.5. Since the address of a bit-field (9.2.4) cannot be taken, a pointer can never point to a bit-field. — *end note*]

8.3.2 References [dcl.ref]

1 In a declaration `T D` where `D` has either of the forms

`&` *attribute-specifier-seq*$_{opt}$ `D1`
`&&` *attribute-specifier-seq*$_{opt}$ `D1`

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is "*derived-declarator-type-list* reference to `T`". The optional *attribute-specifier-seq* appertains to the reference type. Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a *typedef-name* (7.1.3, 14.1) or *decltype-specifier* (7.1.7.2), in which case the cv-qualifiers are ignored. [*Example:*

```
typedef int& A;
const A aref = 3;    // ill-formed; lvalue reference to non-const initialized with rvalue
```

The type of `aref` is "lvalue reference to `int`", not "lvalue reference to `const int`". — *end example*] [*Note:* A reference can be thought of as a name of an object. — *end note*] A declarator that specifies the type "reference to *cv* `void`" is ill-formed.

2 A reference type that is declared using `&` is called an *lvalue reference*, and a reference type that is declared using `&&` is called an *rvalue reference*. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

3 [*Example:*

```
void f(double& a) { a += 3.14; }
// ...
double d = 0;
f(d);
```

declares `a` to be a reference parameter of `f` so the call `f(d)` will add `3.14` to `d`.

```
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```

declares the function `g()` to return a reference to an integer so `g(3)=7` will assign `7` to the fourth element of the array `v`. For another example,

```
struct link {
  link* next;
};

link* first;

void h(link*& p) {  // p is a reference to pointer
  p->next = first;
  first = p;
  p = 0;
}

void k() {
```


4 [*Note:* Forming a pointer to reference type is ill-formed; see 8.3.2. Forming a function pointer type is ill-formed if the function type has *cv-qualifiers* or a *ref-qualifier*; see 8.3.5. Since the address of a bit-field (9.2.4) cannot be taken, a pointer can never point to a bit-field. — *end note*]

8.3.2 References [dcl.ref]

1 In a declaration `T D` where `D` has either of the forms

`&` *attribute-specifier-seq*$_{opt}$ `D1`
`&&` *attribute-specifier-seq*$_{opt}$ `D1`

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is "*derived-declarator-type-list* reference to `T`". The optional *attribute-specifier-seq* appertains to the reference type. Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a *typedef-name* (7.1.3, 14.1) or *decltype-specifier* (7.1.7.2), in which case the cv-qualifiers are ignored. [*Example:*

```
typedef int& A;
const A aref = 3;    // ill-formed; lvalue reference to non-const initialized with rvalue
```

The type of `aref` is "lvalue reference to `int`", not "lvalue reference to `const int`". — *end example*] [*Note:* A reference can be thought of as a name of an object. — *end note*] A declarator that specifies the type "reference to *cv* `void`" is ill-formed.

2 A reference type that is declared using `&` is called an *lvalue reference*, and a reference type that is declared using `&&` is called an *rvalue reference*. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

3 [*Example:*

```
void f(double& a) { a += 3.14; }
// ...
double d = 0;
f(d);
```

declares `a` to be a reference parameter of `f` so the call `f(d)` will add `3.14` to `d`.

```
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```

declares the function `g()` to return a reference to an integer so `g(3)=7` will assign `7` to the fourth element of the array `v`. For another example,

```
struct link {
  link* next;
};

link* first;

void h(link*& p) {  // p is a reference to pointer
  p->next = first;
  first = p;
  p = nullptr;
}

void k() {
```

```
auto fpif(int)->int(*)(int);
```

A *trailing-return-type* is most useful for a type that would be more complicated to specify before the *declarator-id*:

```
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

rather than

```
template <class T, class U> decltype((*(T*)0) + (*(U*)0)) add(T t, U u);
```

— *end note*]

15 A *non-template function* is a function that is not a function template specialization. [*Note:* A function template is not a function. — *end note*]

16 A *declarator-id* or *abstract-declarator* containing an ellipsis shall only be used in a *parameter-declaration*. Such a *parameter-declaration* is a parameter pack (14.5.3). When it is part of a *parameter-declaration-clause*, the parameter pack is a function parameter pack (14.5.3). [*Note:* Otherwise, the *parameter-declaration* is part of a *template-parameter-list* and the parameter pack is a template parameter pack; see 14.1. — *end note*] A function parameter pack is a pack expansion (14.5.3). [*Example:*

```
template<typename... T> void f(T (* ...t)(int, int));

int add(int, int);
float subtract(int, int);

void g() {
  f(add, subtract);
}
```

— *end example*]

17 There is a syntactic ambiguity when an ellipsis occurs at the end of a *parameter-declaration-clause* without a preceding comma. In this case, the ellipsis is parsed as part of the *abstract-declarator* if the type of the parameter either names a template parameter pack that has not been expanded or contains `auto`; otherwise, it is parsed as part of the *parameter-declaration-clause*.[100]

8.3.6 Default arguments [dcl.fct.default]

1 If an *initializer-clause* is specified in a *parameter-declaration* this *initializer-clause* is used as a default argument. Default arguments will be used in calls where trailing arguments are missing.

2 [*Example:* the declaration

```
void point(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type `int`. It can be called in any of these ways:

```
point(1,2);  point(1);  point();
```

The last two calls are equivalent to `point(1,4)` and `point(3,4)`, respectively. — *end example*]

3 A default argument shall be specified only in the *parameter-declaration-clause* of a function declaration or *lambda-declarator* or in a *template-parameter* (14.1); in the latter case, the *initializer-clause* shall be an

100) One can explicitly disambiguate the parse either by introducing a comma (so the ellipsis will be parsed as part of the *parameter-declaration-clause*) or by introducing a name for the parameter (so the ellipsis will be parsed as part of the *declarator-id*).


```
auto fpif(int)->int(*)(int);
```

A *trailing-return-type* is most useful for a type that would be more complicated to specify before the *declarator-id*:

```
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

rather than

```
template <class T, class U> decltype((*(T*)nullptr) + (*(U*)nullptr)) add(T t, U u);
```

— *end note*]

15 A *non-template function* is a function that is not a function template specialization. [*Note:* A function template is not a function. — *end note*]

16 A *declarator-id* or *abstract-declarator* containing an ellipsis shall only be used in a *parameter-declaration*. Such a *parameter-declaration* is a parameter pack (14.5.3). When it is part of a *parameter-declaration-clause*, the parameter pack is a function parameter pack (14.5.3). [*Note:* Otherwise, the *parameter-declaration* is part of a *template-parameter-list* and the parameter pack is a template parameter pack; see 14.1. — *end note*] A function parameter pack is a pack expansion (14.5.3). [*Example:*

```
template<typename... T> void f(T (* ...t)(int, int));

int add(int, int);
float subtract(int, int);

void g() {
  f(add, subtract);
}
```

— *end example*]

17 There is a syntactic ambiguity when an ellipsis occurs at the end of a *parameter-declaration-clause* without a preceding comma. In this case, the ellipsis is parsed as part of the *abstract-declarator* if the type of the parameter either names a template parameter pack that has not been expanded or contains `auto`; otherwise, it is parsed as part of the *parameter-declaration-clause*.[100]

8.3.6 Default arguments [dcl.fct.default]

1 If an *initializer-clause* is specified in a *parameter-declaration* this *initializer-clause* is used as a default argument. Default arguments will be used in calls where trailing arguments are missing.

2 [*Example:* the declaration

```
void point(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type `int`. It can be called in any of these ways:

```
point(1,2);  point(1);  point();
```

The last two calls are equivalent to `point(1,4)` and `point(3,4)`, respectively. — *end example*]

3 A default argument shall be specified only in the *parameter-declaration-clause* of a function declaration or *lambda-declarator* or in a *template-parameter* (14.1); in the latter case, the *initializer-clause* shall be an

100) One can explicitly disambiguate the parse either by introducing a comma (so the ellipsis will be parsed as part of the *parameter-declaration-clause*) or by introducing a name for the parameter (so the ellipsis will be parsed as part of the *declarator-id*).

```
int x;
int y;

struct enclose {
  int x;
  static int s;

  struct inner {
    void f(int i) {
      int a = sizeof(x);        // OK: operand of sizeof is an unevaluated operand
      x = i;                    // error: assign to enclose::x
      s = i;                    // OK: assign to enclose::s
      ::x = i;                  // OK: assign to global x
      y = i;                    // OK: assign to global y
    }
    void g(enclose* p, int i) {
      p->x = i;                 // OK: assign to enclose::x
    }
  };
};

inner* p = 0;                   // error: inner not in scope
```

— *end example*]

2 Member functions and static data members of a nested class can be defined in a namespace scope enclosing the definition of their class. [*Example:*

```
struct enclose {
  struct inner {
    static int x;
    void f(int i);
  };
};

int enclose::inner::x = 1;

void enclose::inner::f(int i) { /* ... */ }
```

— *end example*]

3 If class `X` is defined in a namespace scope, a nested class `Y` may be declared in class `X` and later defined in the definition of class `X` or be later defined in a namespace scope enclosing the definition of class `X`. [*Example:*

```
class E {
  class I1;                     // forward declaration of nested class
  class I2;
  class I1 { };                 // definition of nested class
};
class E::I2 { };                // definition of nested class
```

— *end example*]

4 Like a member function, a friend function (11.3) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (9.2.3), but it has no special access rights to members of an enclosing class.


```
int x;
int y;

struct enclose {
  int x;
  static int s;

  struct inner {
    void f(int i) {
      int a = sizeof(x);        // OK: operand of sizeof is an unevaluated operand
      x = i;                    // error: assign to enclose::x
      s = i;                    // OK: assign to enclose::s
      ::x = i;                  // OK: assign to global x
      y = i;                    // OK: assign to global y
    }
    void g(enclose* p, int i) {
      p->x = i;                 // OK: assign to enclose::x
    }
  };
};

inner* p = nullptr;             // error: inner not in scope
```

— *end example*]

2 Member functions and static data members of a nested class can be defined in a namespace scope enclosing the definition of their class. [*Example:*

```
struct enclose {
  struct inner {
    static int x;
    void f(int i);
  };
};

int enclose::inner::x = 1;

void enclose::inner::f(int i) { /* ... */ }
```

— *end example*]

3 If class `X` is defined in a namespace scope, a nested class `Y` may be declared in class `X` and later defined in the definition of class `X` or be later defined in a namespace scope enclosing the definition of class `X`. [*Example:*

```
class E {
  class I1;                     // forward declaration of nested class
  class I2;
  class I1 { };                 // definition of nested class
};
class E::I2 { };                // definition of nested class
```

— *end example*]

4 Like a member function, a friend function (11.3) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (9.2.3), but it has no special access rights to members of an enclosing class.

```
int x;
void f() {
  static int s ;
  int x;
  const int N = 5;
  extern int q();

  struct local {
    int g() { return x; }       // error: odr-use of automatic variable x
    int h() { return s; }       // OK
    int k() { return ::x; }     // OK
    int l() { return q(); }     // OK
    int m() { return N; }       // OK: not an odr-use
    int* n() { return &N; }     // error: odr-use of automatic variable N
  };
}

local* p = 0;                   // error: local not in scope
```

— *end example*]

2 An enclosing function has no special access to members of the local class; it obeys the usual access rules (Clause 11). Member functions of a local class shall be defined within their class definition, if they are defined at all.

3 If class `X` is a local class a nested class `Y` may be declared in class `X` and later defined in the definition of class `X` or be later defined in the same scope as the definition of class `X`. A class nested within a local class is a local class.

4 A local class shall not have static data members.


```
int x;
void f() {
  static int s ;
  int x;
  const int N = 5;
  extern int q();

  struct local {
    int g() { return x; }       // error: odr-use of automatic variable x
    int h() { return s; }       // OK
    int k() { return ::x; }     // OK
    int l() { return q(); }     // OK
    int m() { return N; }       // OK: not an odr-use
    int* n() { return &N; }     // error: odr-use of automatic variable N
  };
}

local* p = nullptr;             // error: local not in scope
```

— *end example*]

2 An enclosing function has no special access to members of the local class; it obeys the usual access rules (Clause 11). Member functions of a local class shall be defined within their class definition, if they are defined at all.

3 If class `X` is a local class a nested class `Y` may be declared in class `X` and later defined in the definition of class `X` or be later defined in the same scope as the definition of class `X`. A class nested within a local class is a local class.

4 A local class shall not have static data members.

5 A specialization of a conversion function template is referenced in the same way as a non-template conversion function that converts to the same type. [*Example:*

```
struct A {
  template <class T> operator T*();
};
template <class T> A::operator T*(){ return 0; }
template <> A::operator char*(){ return 0; }     // specialization
template A::operator void*();                    // explicit instantiation

int main() {
  A a;
  int* ip;
  ip = a.operator int*();         // explicit call to template operator
                                  // A::operator int*()
}
```

— *end example*] [*Note:* Because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using a function name, there is no way to provide an explicit template argument list for these function templates. — *end note*]

6 A specialization of a conversion function template is not found by name lookup. Instead, any conversion function templates visible in the context of the use are considered. For each such operator, if argument deduction succeeds (14.8.2.3), the resulting specialization is used as if found by name lookup.

7 A *using-declaration* in a derived class cannot refer to a specialization of a conversion function template in a base class.

8 Overload resolution (13.3.3.2) and partial ordering (14.5.6.2) are used to select the best conversion function among multiple specializations of conversion function templates and/or non-template conversion functions.

14.5.3 Variadic templates [temp.variadic]

1 A *template parameter pack* is a template parameter that accepts zero or more template arguments. [*Example:*

```
template<class ... Types> struct Tuple { };

Tuple<> t0;                   // Types contains no arguments
Tuple<int> t1;                // Types contains one argument: int
Tuple<int, float> t2;         // Types contains two arguments: int and float
Tuple<0> error;               // error: 0 is not a type
```

— *end example*]

2 A *function parameter pack* is a function parameter that accepts zero or more function arguments. [*Example:*

```
template<class ... Types> void f(Types ... args);

f();                  // OK: args contains no arguments
f(1);                 // OK: args contains one argument: int
f(2, 1.0);            // OK: args contains two arguments: int and double
```

— *end example*]

3 A *parameter pack* is either a template parameter pack or a function parameter pack.

4 A *pack expansion* consists of a *pattern* and an ellipsis, the instantiation of which produces zero or more instantiations of the pattern in a list (described below). The form of the pattern depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

©ISO/IEC Dxxxx

5 A specialization of a conversion function template is referenced in the same way as a non-template conversion function that converts to the same type. [*Example:*

```
struct A {
  template <class T> operator T*();
};
template <class T> A::operator T*(){ return nullptr; }
template <> A::operator char*(){ return nullptr; }     // specialization
template A::operator void*();                          // explicit instantiation

int main() {
  A a;
  int* ip;
  ip = a.operator int*();         // explicit call to template operator
                                  // A::operator int*()
}
```

— *end example*] [*Note:* Because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using a function name, there is no way to provide an explicit template argument list for these function templates. — *end note*]

6 A specialization of a conversion function template is not found by name lookup. Instead, any conversion function templates visible in the context of the use are considered. For each such operator, if argument deduction succeeds (14.8.2.3), the resulting specialization is used as if found by name lookup.

7 A *using-declaration* in a derived class cannot refer to a specialization of a conversion function template in a base class.

8 Overload resolution (13.3.3.2) and partial ordering (14.5.6.2) are used to select the best conversion function among multiple specializations of conversion function templates and/or non-template conversion functions.

14.5.3 Variadic templates [temp.variadic]

1 A *template parameter pack* is a template parameter that accepts zero or more template arguments. [*Example:*

```
template<class ... Types> struct Tuple { };

Tuple<> t0;                   // Types contains no arguments
Tuple<int> t1;                // Types contains one argument: int
Tuple<int, float> t2;         // Types contains two arguments: int and float
Tuple<0> error;               // error: 0 is not a type
```

— *end example*]

2 A *function parameter pack* is a function parameter that accepts zero or more function arguments. [*Example:*

```
template<class ... Types> void f(Types ... args);

f();                  // OK: args contains no arguments
f(1);                 // OK: args contains one argument: int
f(2, 1.0);            // OK: args contains two arguments: int and double
```

— *end example*]

3 A *parameter pack* is either a template parameter pack or a function parameter pack.

4 A *pack expansion* consists of a *pattern* and an ellipsis, the instantiation of which produces zero or more instantiations of the pattern in a list (described below). The form of the pattern depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

§ 14.5.3 362

```
  A<int> z;
  h(z);             // overload resolution selects h(A<T>&)
  const A<int> z2;
  h(z2);            // h(const T&) is called because h(A<T>&) is not callable
}
```

— end example]

5 [*Note:* Since partial ordering in a call context considers only parameters for which there are explicit call arguments, some parameters are ignored (namely, function parameter packs, parameters with default arguments, and ellipsis parameters). [*Example:*

```
template<class T> void f(T);            // #1
template<class T> void f(T*, int=1);    // #2
template<class T> void g(T);            // #3
template<class T> void g(T*, ...);      // #4

int main() {
  int* ip;
  f(ip);            // calls #2
  g(ip);            // calls #4
}
```

— end example] [*Example:*

```
template<class T, class U> struct A { };

template<class T, class U> void f(U, A<U, T>* p = 0); // #1
template<         class U> void f(U, A<U, U>* p = 0); // #2
template<class T         > void g(T, T = T());       // #3
template<class T, class... U> void g(T, U ...);      // #4

void h() {
  f<int>(42, (A<int, int>*)0);                       // calls #2
  f<int>(42);                                        // error: ambiguous
  g(42);                                             // error: ambiguous
}
```

— end example] [*Example:*

```
template<class T, class... U> void f(T, U...);       // #1
template<class T            > void f(T);             // #2
template<class T, class... U> void g(T*, U...);      // #3
template<class T            > void g(T);             // #4

void h(int i) {
  f(&i);                                             // error: ambiguous
  g(&i);                                             // OK: calls #3
}
```

— end example] *— end note*]

14.5.7 Alias templates [temp.alias]

1 A *template-declaration* in which the *declaration* is an *alias-declaration* (Clause 7) declares the *identifier* to be a *alias template*. An alias template is a name for a family of types. The name of the alias template is a *template-name*.


```
  A<int> z;
  h(z);             // overload resolution selects h(A<T>&)
  const A<int> z2;
  h(z2);            // h(const T&) is called because h(A<T>&) is not callable
}
```

— end example]

5 [*Note:* Since partial ordering in a call context considers only parameters for which there are explicit call arguments, some parameters are ignored (namely, function parameter packs, parameters with default arguments, and ellipsis parameters). [*Example:*

```
template<class T> void f(T);            // #1
template<class T> void f(T*, int=1);    // #2
template<class T> void g(T);            // #3
template<class T> void g(T*, ...);      // #4

int main() {
  int* ip;
  f(ip);            // calls #2
  g(ip);            // calls #4
}
```

— end example] [*Example:*

```
template<class T, class U> struct A { };

template<class T, class U> void f(U, A<U, T>* p = nullptr); // #1
template<         class U> void f(U, A<U, U>* p = nullptr); // #2
template<class T         > void g(T, T = T());             // #3
template<class T, class... U> void g(T, U ...);            // #4

void h() {
  f<int>(42, (A<int, int>*)nullptr);                       // calls #2
  f<int>(42);                                              // error: ambiguous
  g(42);                                                   // error: ambiguous
}
```

— end example] [*Example:*

```
template<class T, class... U> void f(T, U...);       // #1
template<class T            > void f(T);             // #2
template<class T, class... U> void g(T*, U...);      // #3
template<class T            > void g(T);             // #4

void h(int i) {
  f(&i);                                             // error: ambiguous
  g(&i);                                             // OK: calls #3
}
```

— end example] *— end note*]

14.5.7 Alias templates [temp.alias]

1 A *template-declaration* in which the *declaration* is an *alias-declaration* (Clause 7) declares the *identifier* to be a *alias template*. An alias template is a name for a family of types. The name of the alias template is a *template-name*.

does apply to inline functions even though an explicit instantiation declaration of such an entity has no other normative effect. This is needed to ensure that if the address of an inline function is taken in a translation unit in which the implementation chose to suppress the out-of-line body, another translation unit will supply the body. — *end note*] An explicit instantiation declaration shall not name a specialization of a template with internal linkage.

12 The usual access checking rules do not apply to names used to specify explicit instantiations. [*Note:* In particular, the template arguments and names used in the function declarator (including parameter types, return types and exception specifications) may be private types or objects which would normally not be accessible and the template may be a member template or member function which would not normally be accessible. — *end note*]

13 An explicit instantiation does not constitute a use of a default argument, so default argument instantiation is not done. [*Example:*

```
char* p = 0;
template<class T> T g(T x = &p) { return x; }
template int g<int>(int);       // OK even though &p isn't an int.
```

— *end example*]

14.7.3 Explicit specialization [temp.expl.spec]

1 An explicit specialization of any of the following:

(1.1) — function template

(1.2) — class template

(1.3) — variable template

(1.4) — member function of a class template

(1.5) — static data member of a class template

(1.6) — member class of a class template

(1.7) — member enumeration of a class template

(1.8) — member class template of a class or class template

(1.9) — member function template of a class or class template

can be declared by a declaration introduced by `template<>`; that is:

explicit-specialization:
`template < >` *declaration*

[*Example:*

```
template<class T> class stream;

template<> class stream<char> { /* ... */ };

template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

template<> void sort<char*>(Array<char*>&) ;
```


does apply to inline functions even though an explicit instantiation declaration of such an entity has no other normative effect. This is needed to ensure that if the address of an inline function is taken in a translation unit in which the implementation chose to suppress the out-of-line body, another translation unit will supply the body. — *end note*] An explicit instantiation declaration shall not name a specialization of a template with internal linkage.

12 The usual access checking rules do not apply to names used to specify explicit instantiations. [*Note:* In particular, the template arguments and names used in the function declarator (including parameter types, return types and exception specifications) may be private types or objects which would normally not be accessible and the template may be a member template or member function which would not normally be accessible. — *end note*]

13 An explicit instantiation does not constitute a use of a default argument, so default argument instantiation is not done. [*Example:*

```
char* p = nullptr;
template<class T> T g(T x = &p) { return x; }
template int g<int>(int);       // OK even though &p isn't an int.
```

— *end example*]

14.7.3 Explicit specialization [temp.expl.spec]

1 An explicit specialization of any of the following:

(1.1) — function template

(1.2) — class template

(1.3) — variable template

(1.4) — member function of a class template

(1.5) — static data member of a class template

(1.6) — member class of a class template

(1.7) — member enumeration of a class template

(1.8) — member class template of a class or class template

(1.9) — member function template of a class or class template

can be declared by a declaration introduced by `template<>`; that is:

explicit-specialization:
`template < >` *declaration*

[*Example:*

```
template<class T> class stream;

template<> class stream<char> { /* ... */ };

template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

template<> void sort<char*>(Array<char*>&) ;
```

18.2.2 Null pointers [support.types.nullptr]

1 The type `nullptr_t` is a synonym for the type of a `nullptr` expression, and it has the characteristics described in 3.9.1 and 4.11. [*Note:* Although `nullptr`'s address cannot be taken, the address of another `nullptr_t` object that is an lvalue can be taken. — *end note*]

2 The macro `NULL` is an implementation-defined null pointer constant.[190]

18.2.3 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.[191] 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 In general [support.limits.general]

1 The headers `<limits>` (18.3.2), `<climits>` (18.3.3.1), and `<cfloat>` (18.3.3.2) supply characteristics of implementation-dependent arithmetic types (3.9.1).

18.3.2 Numeric limits [limits]

18.3.2.1 Class template `numeric_limits` [limits.numeric]

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

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

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

4 Non-arithmetic standard types, such as `complex<T>` (26.5.2), shall not have specializations.

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

```
namespace std {
  template<class T> class numeric_limits;
  enum float_round_style;
```

190) Possible definitions include `0` and `0L`, but not `(void*)0`.
191) Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.

18.2.2 Null pointers [support.types.nullptr]

1 The type `nullptr_t` is a synonym for the type of a `nullptr` expression, and it has the characteristics described in 3.9.1 and 4.11. [*Note:* Although `nullptr`'s address cannot be taken, the address of another `nullptr_t` object that is an lvalue can be taken. — *end note*]

2 The macro `NULL` is an implementation-defined null pointer constant.[190]

18.2.3 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.[191] 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 In general [support.limits.general]

1 The headers `<limits>` (18.3.2), `<climits>` (18.3.3.1), and `<cfloat>` (18.3.3.2) supply characteristics of implementation-dependent arithmetic types (3.9.1).

18.3.2 Numeric limits [limits]

18.3.2.1 Class template `numeric_limits` [limits.numeric]

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

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

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

4 Non-arithmetic standard types, such as `complex<T>` (26.5.2), shall not have specializations.

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

```
namespace std {
  template<class T> class numeric_limits;
  enum float_round_style;
```

190) Possible definitions include `0` and `0L`, but not `(void*)0`.
191) Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.

```
template<class Y, class D> void reset(Y* p, D d);
```

Effects: Equivalent to `shared_ptr(p, d).swap(*this)`.

```
template<class Y, class D, class A> void reset(Y* p, D d, A a);
```

Effects: Equivalent to `shared_ptr(p, d, a).swap(*this)`.

20.11.2.2.5 shared_ptr observers [util.smartptr.shared.obs]

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

Returns: the stored pointer.

```
T& operator*() const noexcept;
```

Requires: `get() != 0`.

Returns: `*get()`.

Remarks: When `T` is (possibly cv-qualified) `void`, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well formed.

```
T* operator->() const noexcept;
```

Requires: `get() != 0`.

Returns: `get()`.

```
long use_count() const noexcept;
```

Returns: the number of `shared_ptr` objects, `*this` included, that *share ownership* with `*this`, or `0` when `*this` is *empty*.

```
bool unique() const noexcept;
```

Returns: `use_count() == 1`.

[*Note:* If you are using `unique()` to implement copy on write, do not rely on a specific value when `get() == nullptr`. — *end note*]

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

Returns: `get() != 0`.

```
template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;
```

Returns: An unspecified value such that

(11.1) — `x.owner_before(y)` defines a strict weak ordering as defined in 25.5;

(11.2) — under the equivalence relation defined by `owner_before`, `!a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.


```
template<class Y, class D> void reset(Y* p, D d);
```

Effects: Equivalent to `shared_ptr(p, d).swap(*this)`.

```
template<class Y, class D, class A> void reset(Y* p, D d, A a);
```

Effects: Equivalent to `shared_ptr(p, d, a).swap(*this)`.

20.11.2.2.5 shared_ptr observers [util.smartptr.shared.obs]

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

Returns: the stored pointer.

```
T& operator*() const noexcept;
```

Requires: `get() != nullptr`.

Returns: `*get()`.

Remarks: When `T` is (possibly cv-qualified) `void`, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well formed.

```
T* operator->() const noexcept;
```

Requires: `get() != nullptr`.

Returns: `get()`.

```
long use_count() const noexcept;
```

Returns: the number of `shared_ptr` objects, `*this` included, that *share ownership* with `*this`, or `0` when `*this` is *empty*.

```
bool unique() const noexcept;
```

Returns: `use_count() == 1`.

[*Note:* If you are using `unique()` to implement copy on write, do not rely on a specific value when `get() == nullptr`. — *end note*]

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

Returns: `get() != nullptr`.

```
template<class U> bool owner_before(shared_ptr<U> const& b) const;
template<class U> bool owner_before(weak_ptr<U> const& b) const;
```

Returns: An unspecified value such that

(11.1) — `x.owner_before(y)` defines a strict weak ordering as defined in 25.5;

(11.2) — under the equivalence relation defined by `owner_before`, `!a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

20.11.2.2.6 shared_ptr creation [util.smartptr.shared.create]

```
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);
```

1 *Requires:* The expression `::new (pv) T(std::forward<Args>(args)...)`, where `pv` has type `void*` and points to storage suitable to hold an object of type `T`, shall be well formed. `A` shall be an *allocator* (17.6.3.5). The copy constructor and destructor of `A` shall not throw exceptions.

2 *Effects:* Allocates memory suitable for an object of type `T` and constructs an object in that memory via the placement *new-expression* `::new (pv) T(std::forward<Args>(args)...)`. The template `allocate_shared` uses a copy of `a` to allocate memory. If an exception is thrown, the functions have no effect.

3 *Returns:* A `shared_ptr` instance that stores and owns the address of the newly constructed object of type `T`.

4 *Postconditions:* `get() != 0 && use_count() == 1`.

5 *Throws:* `bad_alloc`, or an exception thrown from `A::allocate` or from the constructor of `T`.

6 *Remarks:* Implementations should perform no more than one memory allocation. [*Note:* This provides efficiency equivalent to an intrusive smart pointer. — *end note*]

7 [*Note:* These functions will typically allocate more memory than `sizeof(T)` to allow for internal bookkeeping structures such as the reference counts. — *end note*]

20.11.2.2.7 shared_ptr comparison [util.smartptr.shared.cmp]

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

1 *Returns:* `a.get() == b.get()`.

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

2 *Returns:* `less<V>()(a.get(), b.get())`, where `V` is the composite pointer type (Clause 5) of `T*` and `U*`.

3 [*Note:* Defining a comparison operator allows `shared_ptr` objects to be used as keys in associative containers. — *end note*]

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

4 *Returns:* `!a`.

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

5 *Returns:* `(bool)a`.

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


20.11.2.2.6 shared_ptr creation [util.smartptr.shared.create]

```
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);
```

1 *Requires:* The expression `::new (pv) T(std::forward<Args>(args)...)`, where `pv` has type `void*` and points to storage suitable to hold an object of type `T`, shall be well formed. `A` shall be an *allocator* (17.6.3.5). The copy constructor and destructor of `A` shall not throw exceptions.

2 *Effects:* Allocates memory suitable for an object of type `T` and constructs an object in that memory via the placement *new-expression* `::new (pv) T(std::forward<Args>(args)...)`. The template `allocate_shared` uses a copy of `a` to allocate memory. If an exception is thrown, the functions have no effect.

3 *Returns:* A `shared_ptr` instance that stores and owns the address of the newly constructed object of type `T`.

4 *Postconditions:* `get() != nullptr && use_count() == 1`.

5 *Throws:* `bad_alloc`, or an exception thrown from `A::allocate` or from the constructor of `T`.

6 *Remarks:* Implementations should perform no more than one memory allocation. [*Note:* This provides efficiency equivalent to an intrusive smart pointer. — *end note*]

7 [*Note:* These functions will typically allocate more memory than `sizeof(T)` to allow for internal bookkeeping structures such as the reference counts. — *end note*]

20.11.2.2.7 shared_ptr comparison [util.smartptr.shared.cmp]

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

1 *Returns:* `a.get() == b.get()`.

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

2 *Returns:* `less<V>()(a.get(), b.get())`, where `V` is the composite pointer type (Clause 5) of `T*` and `U*`.

3 [*Note:* Defining a comparison operator allows `shared_ptr` objects to be used as keys in associative containers. — *end note*]

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

4 *Returns:* `!a`.

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

5 *Returns:* `(bool)a`.

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

```
// basic_string typedef names
using string    = basic_string<char>;
using u16string = basic_string<char16_t>;
using u32string = basic_string<char32_t>;
using wstring   = basic_string<wchar_t>;

// 21.3.3, numeric conversions:
int stoi(const string& str, size_t* idx = 0, int base = 10);
long stol(const string& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t* idx = 0, int base = 10);
long long stoll(const string& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = 0, int base = 10);
float stof(const string& str, size_t* idx = 0);
double stod(const string& str, size_t* idx = 0);
long double stold(const string& str, size_t* idx = 0);
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);

int stoi(const wstring& str, size_t* idx = 0, int base = 10);
long stol(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = 0, int base = 10);
long long stoll(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = 0, int base = 10);
float stof(const wstring& str, size_t* idx = 0);
double stod(const wstring& str, size_t* idx = 0);
long double stold(const wstring& str, size_t* idx = 0);
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

// 21.3.4, hash support:
template<class T> struct hash;
template<> struct hash<string>;
template<> struct hash<u16string>;
template<> struct hash<u32string>;
template<> struct hash<wstring>;

namespace pmr {
  template <class charT, class traits = char_traits<charT>>
    using basic_string =
      std::basic_string<charT, traits, polymorphic_allocator<charT>>;
```



```
// basic_string typedef names
using string    = basic_string<char>;
using u16string = basic_string<char16_t>;
using u32string = basic_string<char32_t>;
using wstring   = basic_string<wchar_t>;

// 21.3.3, numeric conversions:
int stoi(const string& str, size_t* idx = nullptr, int base = 10);
long stol(const string& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const string& str, size_t* idx = nullptr, int base = 10);
long long stoll(const string& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = nullptr, int base = 10);
float stof(const string& str, size_t* idx = nullptr);
double stod(const string& str, size_t* idx = nullptr);
long double stold(const string& str, size_t* idx = nullptr);
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);

int stoi(const wstring& str, size_t* idx = nullptr, int base = 10);
long stol(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = nullptr, int base = 10);
long long stoll(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = nullptr, int base = 10);
float stof(const wstring& str, size_t* idx = nullptr);
double stod(const wstring& str, size_t* idx = nullptr);
long double stold(const wstring& str, size_t* idx = nullptr);
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

// 21.3.4, hash support:
template<class T> struct hash;
template<> struct hash<string>;
template<> struct hash<u16string>;
template<> struct hash<u32string>;
template<> struct hash<wstring>;

namespace pmr {
  template <class charT, class traits = char_traits<charT>>
    using basic_string =
      std::basic_string<charT, traits, polymorphic_allocator<charT>>;
```

```
            basic_string<charT, traits, Allocator>& str,
            charT delim);
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
            basic_string<charT, traits, Allocator>& str,
            charT delim);
```

7 *Effects:* Behaves as an unformatted input function (27.7.2.3), except that it does not affect the value returned by subsequent calls to `basic_istream<>::gcount()`. After constructing a `sentry` object, if the sentry converts to true, calls `str.erase()` and then extracts characters from `is` and appends them to `str` as if by calling `str.append(1, c)` until any of the following occurs:

(7.1) — end-of-file occurs on the input sequence (in which case, the `getline` function calls `is.setstate(ios_base::eofbit)`).

(7.2) — `traits::eq(c, delim)` for the next available input character c (in which case, c is extracted but not appended) (27.5.5.4)

(7.3) — `str.max_size()` characters are stored (in which case, the function calls `is.setstate(ios_base::failbit)`) (27.5.5.4)

8 The conditions are tested in the order shown. In any case, after the last character is extracted, the `sentry` object `k` is destroyed.

9 If the function extracts no characters, it calls `is.setstate(ios_base::failbit)` which may throw `ios_base::failure` (27.5.5.4).

10 *Returns:* `is`.

```
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>& is,
            basic_string<charT, traits, Allocator>& str);
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
            basic_string<charT, traits, Allocator>& str);
```

11 *Returns:* `getline(is, str, is.widen('\n'))`

21.3.3 Numeric conversions [string.conversions]

```
int stoi(const string& str, size_t* idx = 0, int base = 10);
long stol(const string& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t* idx = 0, int base = 10);
long long stoll(const string& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = 0, int base = 10);
```

1 *Effects:* the first two functions call `strtol(str.c_str(), ptr, base)`, and the last three functions call `strtoul(str.c_str(), ptr, base)`, `strtoll(str.c_str(), ptr, base)`, and `strtoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

2 *Returns:* The converted result.

3 *Throws:* `invalid_argument` if `strtol`, `strtoul`, `strtoll`, or `strtoull` reports that no conversion could be performed. Throws `out_of_range` if `strtol`, `strtoul`, `strtoll` or `strtoull` sets `errno` to `ERANGE`, or if the converted value is outside the range of representable values for the return type.


```
            basic_string<charT, traits, Allocator>& str,
            charT delim);
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
            basic_string<charT, traits, Allocator>& str,
            charT delim);
```

7 *Effects:* Behaves as an unformatted input function (27.7.2.3), except that it does not affect the value returned by subsequent calls to `basic_istream<>::gcount()`. After constructing a `sentry` object, if the sentry converts to true, calls `str.erase()` and then extracts characters from `is` and appends them to `str` as if by calling `str.append(1, c)` until any of the following occurs:

(7.1) — end-of-file occurs on the input sequence (in which case, the `getline` function calls `is.setstate(ios_base::eofbit)`).

(7.2) — `traits::eq(c, delim)` for the next available input character c (in which case, c is extracted but not appended) (27.5.5.4)

(7.3) — `str.max_size()` characters are stored (in which case, the function calls `is.setstate(ios_base::failbit)`) (27.5.5.4)

8 The conditions are tested in the order shown. In any case, after the last character is extracted, the `sentry` object `k` is destroyed.

9 If the function extracts no characters, it calls `is.setstate(ios_base::failbit)` which may throw `ios_base::failure` (27.5.5.4).

10 *Returns:* `is`.

```
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>& is,
            basic_string<charT, traits, Allocator>& str);
template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>&
    getline(basic_istream<charT, traits>&& is,
            basic_string<charT, traits, Allocator>& str);
```

11 *Returns:* `getline(is, str, is.widen('\n'))`

21.3.3 Numeric conversions [string.conversions]

```
int stoi(const string& str, size_t* idx = nullptr, int base = 10);
long stol(const string& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const string& str, size_t* idx = nullptr, int base = 10);
long long stoll(const string& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = nullptr, int base = 10);
```

1 *Effects:* the first two functions call `strtol(str.c_str(), ptr, base)`, and the last three functions call `strtoul(str.c_str(), ptr, base)`, `strtoll(str.c_str(), ptr, base)`, and `strtoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

2 *Returns:* The converted result.

3 *Throws:* `invalid_argument` if `strtol`, `strtoul`, `strtoll`, or `strtoull` reports that no conversion could be performed. Throws `out_of_range` if `strtol`, `strtoul`, `strtoll` or `strtoull` sets `errno` to `ERANGE`, or if the converted value is outside the range of representable values for the return type.

```
float stof(const string& str, size_t* idx = 0);
double stod(const string& str, size_t* idx = 0);
long double stold(const string& str, size_t* idx = 0);
```

4 *Effects:* These functions call `strtof(str.c_str(), ptr)`, `strtod(str.c_str(), ptr)`, and `strtold(str.c_str(), ptr)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

5 *Returns:* The converted result.

6 *Throws:* `invalid_argument` if `strtof`, `strtod`, or `strtold` reports that no conversion could be performed. Throws `out_of_range` if `strtof`, `strtod`, or `strtold` sets `errno` to `ERANGE` or if the converted value is outside the range of representable values for the return type.

```
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);
```

7 *Returns:* Each function returns a `string` object holding the character representation of the value of its argument that would be generated by calling `sprintf(buf, fmt, val)` with a format specifier of `"%d"`, `"%u"`, `"%ld"`, `"%lu"`, `"%lld"`, `"%llu"`, `"%f"`, `"%f"`, or `"%Lf"`, respectively, where `buf` designates an internal character buffer of sufficient size.

```
int stoi(const wstring& str, size_t* idx = 0, int base = 10);
long stol(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = 0, int base = 10);
long long stoll(const wstring& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = 0, int base = 10);
```

8 *Effects:* the first two functions call `wcstol(str.c_str(), ptr, base)`, and the last three functions call `wcstoul(str.c_str(), ptr, base)`, `wcstoll(str.c_str(), ptr, base)`, and `wcstoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

9 *Returns:* The converted result.

10 *Throws:* `invalid_argument` if `wcstol`, `wcstoul`, `wcstoll`, or `wcstoull` reports that no conversion could be performed. Throws `out_of_range` if the converted value is outside the range of representable values for the return type.

```
float stof(const wstring& str, size_t* idx = 0);
double stod(const wstring& str, size_t* idx = 0);
long double stold(const wstring& str, size_t* idx = 0);
```

11 *Effects:* These functions call `wcstof(str.c_str(), ptr)`, `wcstod(str.c_str(), ptr)`, and `wcstold(str.c_str(), ptr)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store


```
float stof(const string& str, size_t* idx = nullptr);
double stod(const string& str, size_t* idx = nullptr);
long double stold(const string& str, size_t* idx = nullptr);
```

4 *Effects:* These functions call `strtof(str.c_str(), ptr)`, `strtod(str.c_str(), ptr)`, and `strtold(str.c_str(), ptr)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

5 *Returns:* The converted result.

6 *Throws:* `invalid_argument` if `strtof`, `strtod`, or `strtold` reports that no conversion could be performed. Throws `out_of_range` if `strtof`, `strtod`, or `strtold` sets `errno` to `ERANGE` or if the converted value is outside the range of representable values for the return type.

```
string to_string(int val);
string to_string(unsigned val);
string to_string(long val);
string to_string(unsigned long val);
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);
```

7 *Returns:* Each function returns a `string` object holding the character representation of the value of its argument that would be generated by calling `sprintf(buf, fmt, val)` with a format specifier of `"%d"`, `"%u"`, `"%ld"`, `"%lu"`, `"%lld"`, `"%llu"`, `"%f"`, `"%f"`, or `"%Lf"`, respectively, where `buf` designates an internal character buffer of sufficient size.

```
int stoi(const wstring& str, size_t* idx = nullptr, int base = 10);
long stol(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = nullptr, int base = 10);
long long stoll(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = nullptr, int base = 10);
```

8 *Effects:* the first two functions call `wcstol(str.c_str(), ptr, base)`, and the last three functions call `wcstoul(str.c_str(), ptr, base)`, `wcstoll(str.c_str(), ptr, base)`, and `wcstoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

9 *Returns:* The converted result.

10 *Throws:* `invalid_argument` if `wcstol`, `wcstoul`, `wcstoll`, or `wcstoull` reports that no conversion could be performed. Throws `out_of_range` if the converted value is outside the range of representable values for the return type.

```
float stof(const wstring& str, size_t* idx = nullptr);
double stod(const wstring& str, size_t* idx = nullptr);
long double stold(const wstring& str, size_t* idx = nullptr);
```

11 *Effects:* These functions call `wcstof(str.c_str(), ptr)`, `wcstod(str.c_str(), ptr)`, and `wcstold(str.c_str(), ptr)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store

22.3.3.2.3 Buffer conversions [conversions.buffer]

1 Class template `wbuffer_convert` looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template `wstring_convert`, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

2 **Class template `wbuffer_convert` synopsis**

```
namespace std {
template<class Codecvt,
    class Elem = wchar_t,
    class Tr = std::char_traits<Elem> >
  class wbuffer_convert
    : public std::basic_streambuf<Elem, Tr> {
  public:
    using state_type = typename Codecvt::state_type;

    explicit wbuffer_convert(std::streambuf* bytebuf = 0,
                             Codecvt* pcvt = new Codecvt,
                             state_type state = state_type());

    ~wbuffer_convert();

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

    std::streambuf* rdbuf() const;
    std::streambuf* rdbuf(std::streambuf* bytebuf);

    state_type state() const;

  private:
    std::streambuf* bufptr;          // exposition only
    Codecvt* cvtptr;                 // exposition only
    state_type cvtstate;             // exposition only
  };
}
```

3 The class template describes a stream buffer that controls the transmission of elements of type `Elem`, whose character traits are described by the class `Tr`, to and from a byte stream buffer of type `std::streambuf`. Conversion between a sequence of `Elem` values and multibyte sequences is performed by an object of class `Codecvt`, which shall meet the requirements of the standard code-conversion facet `std::codecvt<Elem, char, std::mbstate_t>`.

4 An object of this class template stores:

(4.1) — `bufptr` — a pointer to its underlying byte stream buffer

(4.2) — `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wbuffer_convert` object is destroyed)

(4.3) — `cvtstate` — a conversion state object

```
state_type state() const;
```

5 *Returns:* `cvtstate`.

```
std::streambuf* rdbuf() const;
```


22.3.3.2.3 Buffer conversions [conversions.buffer]

1 Class template `wbuffer_convert` looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template `wstring_convert`, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

2 **Class template `wbuffer_convert` synopsis**

```
namespace std {
template<class Codecvt,
    class Elem = wchar_t,
    class Tr = std::char_traits<Elem> >
  class wbuffer_convert
    : public std::basic_streambuf<Elem, Tr> {
  public:
    using state_type = typename Codecvt::state_type;

    explicit wbuffer_convert(std::streambuf* bytebuf = nullptr,
                             Codecvt* pcvt = new Codecvt,
                             state_type state = state_type());

    ~wbuffer_convert();

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

    std::streambuf* rdbuf() const;
    std::streambuf* rdbuf(std::streambuf* bytebuf);

    state_type state() const;

  private:
    std::streambuf* bufptr;          // exposition only
    Codecvt* cvtptr;                 // exposition only
    state_type cvtstate;             // exposition only
  };
}
```

3 The class template describes a stream buffer that controls the transmission of elements of type `Elem`, whose character traits are described by the class `Tr`, to and from a byte stream buffer of type `std::streambuf`. Conversion between a sequence of `Elem` values and multibyte sequences is performed by an object of class `Codecvt`, which shall meet the requirements of the standard code-conversion facet `std::codecvt<Elem, char, std::mbstate_t>`.

4 An object of this class template stores:

(4.1) — `bufptr` — a pointer to its underlying byte stream buffer

(4.2) — `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wbuffer_convert` object is destroyed)

(4.3) — `cvtstate` — a conversion state object

```
state_type state() const;
```

5 *Returns:* `cvtstate`.

```
std::streambuf* rdbuf() const;
```

6 *Returns:* `bufptr`.

```
std::streambuf* rdbuf(std::streambuf* bytebuf);
```

7 *Effects:* Stores `bytebuf` in `bufptr`.

8 *Returns:* The previous value of `bufptr`.

```
using state_type = typename Codecvt::state_type;
```

9 The type shall be a synonym for `Codecvt::state_type`.

```
explicit wbuffer_convert(std::streambuf* bytebuf = 0,
    Codecvt* pcvt = new Codecvt, state_type state = state_type());
```

10 *Requires:* `pcvt != nullptr`.

11 *Effects:* The constructor constructs a stream buffer object, initializes `bufptr` to `bytebuf`, initializes `cvtptr` to `pcvt`, and initializes `cvtstate` to `state`.

```
~wbuffer_convert();
```

12 *Effects:* The destructor shall delete `cvtptr`.

22.4 Standard `locale` categories [locale.categories]

1 Each of the standard categories includes a family of facets. Some of these implement formatting or parsing of a datum, for use by standard or users' iostream operators `<<` and `>>`, as members `put()` and `get()`, respectively. Each such member function takes an `ios_base&` argument whose members `flags()`, `precision()`, and `width()`, specify the format of the corresponding datum (27.5.3). Those functions which need to use other facets call its member `getloc()` to retrieve the locale imbued there. Formatting facets use the character argument `fill` to fill out the specified width where necessary.

2 The `put()` members make no provision for error reporting. (Any failures of the OutputIterator argument must be extracted from the returned iterator.) The `get()` members take an `ios_base::iostate&` argument whose value they ignore, but set to `ios_base::failbit` in case of a parse error.

3 Within this clause it is unspecified whether one virtual function calls another virtual function.

22.4.1 The `ctype` category [category.ctype]

```
namespace std {
  class ctype_base {
  public:
    using mask = T;

    // numeric values are for exposition only.
    static const mask space = 1 << 0;
    static const mask print = 1 << 1;
    static const mask cntrl = 1 << 2;
    static const mask upper = 1 << 3;
    static const mask lower = 1 << 4;
    static const mask alpha = 1 << 5;
    static const mask digit = 1 << 6;
    static const mask punct = 1 << 7;
    static const mask xdigit = 1 << 8;
    static const mask blank = 1 << 9;
    static const mask alnum = alpha | digit;
    static const mask graph = alnum | punct;
  };
}
```


©ISO/IEC Dxxxx

6 *Returns:* `bufptr`.

```
std::streambuf* rdbuf(std::streambuf* bytebuf);
```

7 *Effects:* Stores `bytebuf` in `bufptr`.

8 *Returns:* The previous value of `bufptr`.

```
using state_type = typename Codecvt::state_type;
```

9 The type shall be a synonym for `Codecvt::state_type`.

```
explicit wbuffer_convert(std::streambuf* bytebuf = nullptr,
    Codecvt* pcvt = new Codecvt, state_type state = state_type());
```

10 *Requires:* `pcvt != nullptr`.

11 *Effects:* The constructor constructs a stream buffer object, initializes `bufptr` to `bytebuf`, initializes `cvtptr` to `pcvt`, and initializes `cvtstate` to `state`.

```
~wbuffer_convert();
```

12 *Effects:* The destructor shall delete `cvtptr`.

22.4 Standard `locale` categories [locale.categories]

1 Each of the standard categories includes a family of facets. Some of these implement formatting or parsing of a datum, for use by standard or users' iostream operators `<<` and `>>`, as members `put()` and `get()`, respectively. Each such member function takes an `ios_base&` argument whose members `flags()`, `precision()`, and `width()`, specify the format of the corresponding datum (27.5.3). Those functions which need to use other facets call its member `getloc()` to retrieve the locale imbued there. Formatting facets use the character argument `fill` to fill out the specified width where necessary.

2 The `put()` members make no provision for error reporting. (Any failures of the OutputIterator argument must be extracted from the returned iterator.) The `get()` members take an `ios_base::iostate&` argument whose value they ignore, but set to `ios_base::failbit` in case of a parse error.

3 Within this clause it is unspecified whether one virtual function calls another virtual function.

22.4.1 The `ctype` category [category.ctype]

```
namespace std {
  class ctype_base {
  public:
    using mask = T;

    // numeric values are for exposition only.
    static const mask space = 1 << 0;
    static const mask print = 1 << 1;
    static const mask cntrl = 1 << 2;
    static const mask upper = 1 << 3;
    static const mask lower = 1 << 4;
    static const mask alpha = 1 << 5;
    static const mask digit = 1 << 6;
    static const mask punct = 1 << 7;
    static const mask xdigit = 1 << 8;
    static const mask blank = 1 << 9;
    static const mask alnum = alpha | digit;
    static const mask graph = alnum | punct;
  };
}
```

§ 22.4.1 792

is `true` (unless `do_narrow` returns `dfault`). In addition, for any digit character `c`, the expression `(do_narrow(c, dfault) - '0')` evaluates to the digit value of the character. The second form transforms each character `*p` in the range `[low, high)`, placing the result (or `dfault` if no simple transformation is readily available) in `dest[p-low]`.

14 *Returns:* The first form returns the transformed value; or `dfault` if no mapping is readily available. The second form returns `high`.

22.4.1.2 Class template ctype_byname [locale.ctype.byname]

```
namespace std {
  template <class charT>
  class ctype_byname : public ctype<charT> {
  public:
    using mask = typename ctype<charT>::mask;
    explicit ctype_byname(const char*, size_t refs = 0);
    explicit ctype_byname(const string&, size_t refs = 0);
  protected:
   ~ctype_byname();
  };
}
```

22.4.1.3 ctype specializations [facet.ctype.special]

```
namespace std {
  template <> class ctype<char>
    : public locale::facet, public ctype_base {
  public:
    using char_type = char;

    explicit ctype(const mask* tab = 0, bool del = false,
                   size_t refs = 0);

    bool is(mask m, char c) const;
    const char* is(const char* low, const char* high, mask* vec) const;
    const char* scan_is (mask m,
                         const char* low, const char* high) const;
    const char* scan_not(mask m,
                         const char* low, const char* high) const;

    char        toupper(char c) const;
    const char* toupper(char* low, const char* high) const;
    char        tolower(char c) const;
    const char* tolower(char* low, const char* high) const;

    char  widen(char c) const;
    const char* widen(const char* low, const char* high, char* to) const;
    char  narrow(char c, char dfault) const;
    const char* narrow(const char* low, const char* high, char dfault,
                       char* to) const;

    static locale::id id;
    static const size_t table_size = implementation-defined;

    const mask* table() const noexcept;
    static const mask* classic_table() noexcept;
```


is `true` (unless `do_narrow` returns `dfault`). In addition, for any digit character `c`, the expression `(do_narrow(c, dfault) - '0')` evaluates to the digit value of the character. The second form transforms each character `*p` in the range `[low, high)`, placing the result (or `dfault` if no simple transformation is readily available) in `dest[p-low]`.

14 *Returns:* The first form returns the transformed value; or `dfault` if no mapping is readily available. The second form returns `high`.

22.4.1.2 Class template ctype_byname [locale.ctype.byname]

```
namespace std {
  template <class charT>
  class ctype_byname : public ctype<charT> {
  public:
    using mask = typename ctype<charT>::mask;
    explicit ctype_byname(const char*, size_t refs = 0);
    explicit ctype_byname(const string&, size_t refs = 0);
  protected:
   ~ctype_byname();
  };
}
```

22.4.1.3 ctype specializations [facet.ctype.special]

```
namespace std {
  template <> class ctype<char>
    : public locale::facet, public ctype_base {
  public:
    using char_type = char;

    explicit ctype(const mask* tab = nullptr, bool del = false,
                   size_t refs = 0);

    bool is(mask m, char c) const;
    const char* is(const char* low, const char* high, mask* vec) const;
    const char* scan_is (mask m,
                         const char* low, const char* high) const;
    const char* scan_not(mask m,
                         const char* low, const char* high) const;

    char        toupper(char c) const;
    const char* toupper(char* low, const char* high) const;
    char        tolower(char c) const;
    const char* tolower(char* low, const char* high) const;

    char  widen(char c) const;
    const char* widen(const char* low, const char* high, char* to) const;
    char  narrow(char c, char dfault) const;
    const char* narrow(const char* low, const char* high, char dfault,
                       char* to) const;

    static locale::id id;
    static const size_t table_size = implementation-defined;

    const mask* table() const noexcept;
    static const mask* classic_table() noexcept;
```

```
  protected:
   ~ctype();
    virtual char        do_toupper(char c) const;
    virtual const char* do_toupper(char* low, const char* high) const;
    virtual char        do_tolower(char c) const;
    virtual const char* do_tolower(char* low, const char* high) const;

    virtual char        do_widen(char c) const;
    virtual const char* do_widen(const char* low,
                                 const char* high,
                                 char* to) const;
    virtual char        do_narrow(char c, char dfault) const;
    virtual const char* do_narrow(const char* low,
                                  const char* high,
                                  char dfault, char* to) const;
  };
}
```

1 A specialization `ctype<char>` is provided so that the member functions on type `char` can be implemented `inline`.[238] The implementation-defined value of member `table_size` is at least 256.

22.4.1.3.1 ctype<char> destructor [facet.ctype.char.dtor]

```
~ctype();
```

1 *Effects:* If the constructor's first argument was nonzero, and its second argument was true, does `delete [] table()`.

22.4.1.3.2 ctype<char> members [facet.ctype.char.members]

1 In the following member descriptions, for `unsigned char` values `v` where `v >= table_size`, `table()[v]` is assumed to have an implementation-specific value (possibly different for each such value `v`) without performing the array lookup.

```
explicit ctype(const mask* tbl = 0, bool del = false,
               size_t refs = 0);
```

2 *Requires:* `tbl` either 0 or an array of at least `table_size` elements.

3 *Effects:* Passes its `refs` argument to its base class constructor.

```
bool        is(mask m, char c) const;
const char* is(const char* low, const char* high,
               mask* vec) const;
```

4 *Effects:* The second form, for all `*p` in the range `[low, high)`, assigns into `vec[p-low]` the value `table()[(unsigned char)*p]`.

5 *Returns:* The first form returns `table()[(unsigned char)c] & m`; the second form returns `high`.

```
const char* scan_is(mask m,
                    const char* low, const char* high) const;
```

6 *Returns:* The smallest `p` in the range `[low, high)` such that

```
  table()[(unsigned char) *p] & m
```

is `true`.

238) Only the `char` (not `unsigned char` and `signed char`) form is provided. The specialization is specified in the standard, and not left as an implementation detail, because it affects the derivation interface for `ctype<char>`.


```
  protected:
   ~ctype();
    virtual char        do_toupper(char c) const;
    virtual const char* do_toupper(char* low, const char* high) const;
    virtual char        do_tolower(char c) const;
    virtual const char* do_tolower(char* low, const char* high) const;

    virtual char        do_widen(char c) const;
    virtual const char* do_widen(const char* low,
                                 const char* high,
                                 char* to) const;
    virtual char        do_narrow(char c, char dfault) const;
    virtual const char* do_narrow(const char* low,
                                  const char* high,
                                  char dfault, char* to) const;
  };
}
```

1 A specialization `ctype<char>` is provided so that the member functions on type `char` can be implemented `inline`.[238] The implementation-defined value of member `table_size` is at least 256.

22.4.1.3.1 ctype<char> destructor [facet.ctype.char.dtor]

```
~ctype();
```

1 *Effects:* If the constructor's first argument was nonzero, and its second argument was true, does `delete [] table()`.

22.4.1.3.2 ctype<char> members [facet.ctype.char.members]

1 In the following member descriptions, for `unsigned char` values `v` where `v >= table_size`, `table()[v]` is assumed to have an implementation-specific value (possibly different for each such value `v`) without performing the array lookup.

```
explicit ctype(const mask* tbl = nullptr, bool del = false,
               size_t refs = 0);
```

2 *Requires:* `tbl` either 0 or an array of at least `table_size` elements.

3 *Effects:* Passes its `refs` argument to its base class constructor.

```
bool        is(mask m, char c) const;
const char* is(const char* low, const char* high,
               mask* vec) const;
```

4 *Effects:* The second form, for all `*p` in the range `[low, high)`, assigns into `vec[p-low]` the value `table()[(unsigned char)*p]`.

5 *Returns:* The first form returns `table()[(unsigned char)c] & m`; the second form returns `high`.

```
const char* scan_is(mask m,
                    const char* low, const char* high) const;
```

6 *Returns:* The smallest `p` in the range `[low, high)` such that

```
  table()[(unsigned char) *p] & m
```

is `true`.

238) Only the `char` (not `unsigned char` and `signed char`) form is provided. The specialization is specified in the standard, and not left as an implementation detail, because it affects the derivation interface for `ctype<char>`.

```
  template <class T, class charT, class traits, class Distance>
    bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
                    const istream_iterator<T,charT,traits,Distance>& y);
  template <class T, class charT, class traits, class Distance>
    bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
                    const istream_iterator<T,charT,traits,Distance>& y);
}
```

24.6.1.1 istream_iterator constructors and destructor [istream.iterator.cons]

`see below istream_iterator();`

1 *Effects:* Constructs the end-of-stream iterator. If `T` is a literal type, then this constructor shall be a `constexpr` constructor.

2 *Postcondition:* `in_stream == 0`.

`istream_iterator(istream_type& s);`

3 *Effects:* Initializes *in_stream* with `addressof(s)`. *value* may be initialized during construction or the first time it is referenced.

4 *Postcondition:* `in_stream == addressof(s)`.

`istream_iterator(const istream_iterator& x) = default;`

5 *Effects:* Constructs a copy of `x`. If `T` is a literal type, then this constructor shall be a trivial copy constructor.

6 *Postcondition:* `in_stream == x.in_stream`.

`~istream_iterator() = default;`

7 *Effects:* The iterator is destroyed. If `T` is a literal type, then this destructor shall be a trivial destructor.

24.6.1.2 istream_iterator operations [istream.iterator.ops]

`const T& operator*() const;`

1 *Returns:* *value*.

`const T* operator->() const;`

2 *Returns:* `addressof(operator*())`.

`istream_iterator& operator++();`

3 *Requires:* `in_stream != 0`.

4 *Effects:* As if by: `*in_stream >>value;`

5 *Returns:* `*this`.

`istream_iterator operator++(int);`

6 *Requires:* `in_stream != 0`.

7 *Effects:* As if by:

```
istream_iterator tmp = *this;
*in_stream >> value;
return (tmp);
```



```
  template <class T, class charT, class traits, class Distance>
    bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
                    const istream_iterator<T,charT,traits,Distance>& y);
  template <class T, class charT, class traits, class Distance>
    bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
                    const istream_iterator<T,charT,traits,Distance>& y);
}
```

24.6.1.1 istream_iterator constructors and destructor [istream.iterator.cons]

`see below istream_iterator();`

1 *Effects:* Constructs the end-of-stream iterator. If `T` is a literal type, then this constructor shall be a `constexpr` constructor.

2 *Postcondition:* `in_stream == nullptr`.

`istream_iterator(istream_type& s);`

3 *Effects:* Initializes *in_stream* with `addressof(s)`. *value* may be initialized during construction or the first time it is referenced.

4 *Postcondition:* `in_stream == addressof(s)`.

`istream_iterator(const istream_iterator& x) = default;`

5 *Effects:* Constructs a copy of `x`. If `T` is a literal type, then this constructor shall be a trivial copy constructor.

6 *Postcondition:* `in_stream == x.in_stream`.

`~istream_iterator() = default;`

7 *Effects:* The iterator is destroyed. If `T` is a literal type, then this destructor shall be a trivial destructor.

24.6.1.2 istream_iterator operations [istream.iterator.ops]

`const T& operator*() const;`

1 *Returns:* *value*.

`const T* operator->() const;`

2 *Returns:* `addressof(operator*())`.

`istream_iterator& operator++();`

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

4 *Effects:* As if by: `*in_stream >>value;`

5 *Returns:* `*this`.

`istream_iterator operator++(int);`

6 *Requires:* `in_stream != nullptr`.

7 *Effects:* As if by:

```
istream_iterator tmp = *this;
*in_stream >> value;
return (tmp);
```

```
void*& pword(int idx);
```

Effects: If `parray` is a null pointer, allocates an array of pointers to `void` of unspecified size and stores a pointer to its first element in `parray`. The function then extends the array pointed at by `parray` as necessary to include the element `parray[idx]`. Each newly allocated element of the array is initialized to a null pointer. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to `copyfmt`, calling `pword` with the same index yields another reference to the same value. If the function fails[300] and `*this` is a base subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

Returns: On success `parray[idx]`. On failure a valid `void*&` initialized to 0.

Remarks: After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

27.5.3.6 `ios_base` callbacks [ios.base.callback]

```
void register_callback(event_callback fn, int index);
```

Effects: Registers the pair `(fn, index)` such that during calls to `imbue()` (27.5.3.3), `copyfmt()`, or `~ios_base()` (27.5.3.7), the function `fn` is called with argument `index`. Functions registered are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event.

Requires: The function `fn` shall not throw exceptions.

Remarks: Identical pairs are not merged. A function registered twice will be called twice.

27.5.3.7 `ios_base` constructors/destructor [ios.base.cons]

```
ios_base();
```

Effects: Each `ios_base` member has an indeterminate value after construction. The object's members shall be initialized by calling `basic_ios::init` before the object's first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

```
~ios_base();
```

Effects: Destroys an object of class `ios_base`. Calls each registered callback pair `(fn, index)` (27.5.3.6) as `(*fn)(erase_event, *this, index)` at such time that any `ios_base` member function called from within `fn` has well defined results.

27.5.4 Class template `fpos` [fpos]

```
namespace std {
  template <class stateT> class fpos {
  public:
    // 27.5.4.1, members:
    stateT state() const;
    void state(stateT);
  private;
    stateT st; // exposition only
  };
}
```

27.5.4.1 `fpos` members [fpos.members]

300) for example, because it cannot allocate space.


```
void*& pword(int idx);
```

Effects: If `parray` is a null pointer, allocates an array of pointers to `void` of unspecified size and stores a pointer to its first element in `parray`. The function then extends the array pointed at by `parray` as necessary to include the element `parray[idx]`. Each newly allocated element of the array is initialized to a null pointer. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to `copyfmt`, calling `pword` with the same index yields another reference to the same value. If the function fails[300] and `*this` is a base subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

Returns: On success `parray[idx]`. On failure a valid `void*&` initialized to `nullptr`.

Remarks: After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

27.5.3.6 `ios_base` callbacks [ios.base.callback]

```
void register_callback(event_callback fn, int index);
```

Effects: Registers the pair `(fn, index)` such that during calls to `imbue()` (27.5.3.3), `copyfmt()`, or `~ios_base()` (27.5.3.7), the function `fn` is called with argument `index`. Functions registered are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event.

Requires: The function `fn` shall not throw exceptions.

Remarks: Identical pairs are not merged. A function registered twice will be called twice.

27.5.3.7 `ios_base` constructors/destructor [ios.base.cons]

```
ios_base();
```

Effects: Each `ios_base` member has an indeterminate value after construction. The object's members shall be initialized by calling `basic_ios::init` before the object's first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

```
~ios_base();
```

Effects: Destroys an object of class `ios_base`. Calls each registered callback pair `(fn, index)` (27.5.3.6) as `(*fn)(erase_event, *this, index)` at such time that any `ios_base` member function called from within `fn` has well defined results.

27.5.4 Class template `fpos` [fpos]

```
namespace std {
  template <class stateT> class fpos {
  public:
    // 27.5.4.1, members:
    stateT state() const;
    void state(stateT);
  private;
    stateT st; // exposition only
  };
}
```

27.5.4.1 `fpos` members [fpos.members]

300) for example, because it cannot allocate space.

```
  };
}
```

27.5.5.2 basic_ios constructors [basic.ios.cons]

```
explicit basic_ios(basic_streambuf<charT, traits>* sb);
```

1 *Effects:* Constructs an object of class `basic_ios`, assigning initial values to its member objects by calling `init(sb)`.

```
basic_ios();
```

2 *Effects:* Constructs an object of class `basic_ios` (27.5.3.7) leaving its member objects uninitialized. The object shall be initialized by calling `basic_ios::init` before its first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

```
~basic_ios();
```

3 *Remarks:* The destructor does not destroy `rdbuf()`.

```
void init(basic_streambuf<charT, traits>* sb);
```

Postconditions: The postconditions of this function are indicated in Table 109.

Table 109 — `basic_ios::init()` effects

Element	Value
`rdbuf()`	`sb`
`tie()`	`0`
`rdstate()`	`goodbit` if `sb` is not a null pointer, otherwise `badbit`.
`exceptions()`	`goodbit`
`flags()`	`skipws | dec`
`width()`	`0`
`precision()`	`6`
`fill()`	`widen(' ');`
`getloc()`	a copy of the value returned by `locale()`
`iarray`	a null pointer
`parray`	a null pointer

27.5.5.3 Member functions [basic.ios.members]

```
basic_ostream<charT, traits>* tie() const;
```

1 *Returns:* An output sequence that is *tied* to (synchronized with) the sequence controlled by the stream buffer.

```
basic_ostream<charT, traits>* tie(basic_ostream<charT, traits>* tiestr);
```

2 *Requires:* If `tiestr` is not null, `tiestr` must not be reachable by traversing the linked list of tied stream objects starting from `tiestr->tie()`.

3 *Postcondition:* `tiestr == tie()`.

4 *Returns:* The previous value of `tie()`.

```
basic_streambuf<charT, traits>* rdbuf() const;
```



```
  };
}
```

27.5.5.2 basic_ios constructors [basic.ios.cons]

```
explicit basic_ios(basic_streambuf<charT, traits>* sb);
```

1 *Effects:* Constructs an object of class `basic_ios`, assigning initial values to its member objects by calling `init(sb)`.

```
basic_ios();
```

2 *Effects:* Constructs an object of class `basic_ios` (27.5.3.7) leaving its member objects uninitialized. The object shall be initialized by calling `basic_ios::init` before its first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

```
~basic_ios();
```

3 *Remarks:* The destructor does not destroy `rdbuf()`.

```
void init(basic_streambuf<charT, traits>* sb);
```

Postconditions: The postconditions of this function are indicated in Table 109.

Table 109 — `basic_ios::init()` effects

Element	Value
`rdbuf()`	`sb`
`tie()`	`nullptr`
`rdstate()`	`goodbit` if `sb` is not a null pointer, otherwise `badbit`.
`exceptions()`	`goodbit`
`flags()`	`skipws | dec`
`width()`	`0`
`precision()`	`6`
`fill()`	`widen(' ');`
`getloc()`	a copy of the value returned by `locale()`
`iarray`	a null pointer
`parray`	a null pointer

27.5.5.3 Member functions [basic.ios.members]

```
basic_ostream<charT, traits>* tie() const;
```

1 *Returns:* An output sequence that is *tied* to (synchronized with) the sequence controlled by the stream buffer.

```
basic_ostream<charT, traits>* tie(basic_ostream<charT, traits>* tiestr);
```

2 *Requires:* If `tiestr` is not null, `tiestr` must not be reachable by traversing the linked list of tied stream objects starting from `tiestr->tie()`.

3 *Postcondition:* `tiestr == tie()`.

4 *Returns:* The previous value of `tie()`.

```
basic_streambuf<charT, traits>* rdbuf() const;
```

Returns: A pointer to the `streambuf` associated with the stream.

```
basic_streambuf<charT, traits>* rdbuf(basic_streambuf<charT, traits>* sb);
```

Postcondition: `sb == rdbuf()`.

Effects: Calls `clear()`.

Returns: The previous value of `rdbuf()`.

```
locale imbue(const locale& loc);
```

Effects: Calls `ios_base::imbue(loc)` (27.5.3.3) and if `rdbuf() != 0` then `rdbuf()->pubimbue(loc)` (27.6.3.2.1).

Returns: The prior value of `ios_base::imbue()`.

```
char narrow(char_type c, char dfault) const;
```

Returns: `use_facet< ctype<char_type> >(getloc()).narrow(c, dfault)`

```
char_type widen(char c) const;
```

Returns: `use_facet< ctype<char_type> >(getloc()).widen(c)`

```
char_type fill() const;
```

Returns: The character used to pad (fill) an output conversion to the specified field width.

```
char_type fill(char_type fillch);
```

Postcondition: `traits::eq(fillch, fill())`

Returns: The previous value of `fill()`.

```
basic_ios& copyfmt(const basic_ios& rhs);
```

Effects: If `(this == &rhs)` does nothing. Otherwise assigns to the member objects of `*this` the corresponding member objects of `rhs` as follows:

1. calls each registered callback pair `(fn, index)` as `(*fn)(erase_event, *this, index)`;
2. assigns to the member objects of `*this` the corresponding member objects of `rhs`, except that
   - (16.1) — `rdstate()`, `rdbuf()`, and `exceptions()` are left unchanged;
   - (16.2) — the contents of arrays pointed at by `pword` and `iword` are copied, not the pointers themselves;[301] and
   - (16.3) — if any newly stored pointer values in `*this` point at objects stored outside the object `rhs` and those objects are destroyed when `rhs` is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects;
3. calls each callback pair that was copied from `rhs` as `(*fn)(copyfmt_event, *this, index)`;
4. calls `exceptions(rhs.exceptions())`.

Note: The second pass through the callback pairs permits a copied `pword` value to be zeroed, or to have its referent deep copied or reference counted, or to have other special action taken.

Postconditions: The postconditions of this function are indicated in Table 110.

Returns: `*this`.

301) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.

©ISO/IEC Dxxxx

Returns: A pointer to the `streambuf` associated with the stream.

```
basic_streambuf<charT, traits>* rdbuf(basic_streambuf<charT, traits>* sb);
```

Postcondition: `sb == rdbuf()`.

Effects: Calls `clear()`.

Returns: The previous value of `rdbuf()`.

```
locale imbue(const locale& loc);
```

Effects: Calls `ios_base::imbue(loc)` (27.5.3.3) and if `rdbuf() != nullptr` then `rdbuf()->pubimbue(loc)` (27.6.3.2.1).

Returns: The prior value of `ios_base::imbue()`.

```
char narrow(char_type c, char dfault) const;
```

Returns: `use_facet< ctype<char_type> >(getloc()).narrow(c, dfault)`

```
char_type widen(char c) const;
```

Returns: `use_facet< ctype<char_type> >(getloc()).widen(c)`

```
char_type fill() const;
```

Returns: The character used to pad (fill) an output conversion to the specified field width.

```
char_type fill(char_type fillch);
```

Postcondition: `traits::eq(fillch, fill())`

Returns: The previous value of `fill()`.

```
basic_ios& copyfmt(const basic_ios& rhs);
```

Effects: If `(this == &rhs)` does nothing. Otherwise assigns to the member objects of `*this` the corresponding member objects of `rhs` as follows:

1. calls each registered callback pair `(fn, index)` as `(*fn)(erase_event, *this, index)`;
2. assigns to the member objects of `*this` the corresponding member objects of `rhs`, except that
   - (16.1) — `rdstate()`, `rdbuf()`, and `exceptions()` are left unchanged;
   - (16.2) — the contents of arrays pointed at by `pword` and `iword` are copied, not the pointers themselves;[301] and
   - (16.3) — if any newly stored pointer values in `*this` point at objects stored outside the object `rhs` and those objects are destroyed when `rhs` is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects;
3. calls each callback pair that was copied from `rhs` as `(*fn)(copyfmt_event, *this, index)`;
4. calls `exceptions(rhs.exceptions())`.

Note: The second pass through the callback pairs permits a copied `pword` value to be zeroed, or to have its referent deep copied or reference counted, or to have other special action taken.

Postconditions: The postconditions of this function are indicated in Table 110.

Returns: `*this`.

301) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.

§ 27.5.5.3 1159

Table 110 — `basic_ios::copyfmt()` effects

Element	Value
`rdbuf()`	*unchanged*
`tie()`	`rhs.tie()`
`rdstate()`	*unchanged*
`exceptions()`	`rhs.exceptions()`
`flags()`	`rhs.flags()`
`width()`	`rhs.width()`
`precision()`	`rhs.precision()`
`fill()`	`rhs.fill()`
`getloc()`	`rhs.getloc()`

```
void move(basic_ios& rhs);
void move(basic_ios&& rhs);
```

20 *Postconditions:* `*this` shall have the state that `rhs` had before the function call, except that `rdbuf()` shall return `0`. `rhs` shall be in a valid but unspecified state, except that `rhs.rdbuf()` shall return the same value as it returned before the function call, and `rhs.tie()` shall return `0`.

```
void swap(basic_ios& rhs) noexcept;
```

21 *Effects:* The states of `*this` and `rhs` shall be exchanged, except that `rdbuf()` shall return the same value as it returned before the function call, and `rhs.rdbuf()` shall return the same value as it returned before the function call.

```
void set_rdbuf(basic_streambuf<charT, traits>* sb);
```

22 *Requires:* `sb != nullptr`.

23 *Effects:* Associates the `basic_streambuf` object pointed to by `sb` with this stream without calling `clear()`.

24 *Postconditions:* `rdbuf() == sb`.

25 *Throws:* Nothing.

27.5.5.4 `basic_ios` flags functions [iostate.flags]

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

1 *Returns:* `!fail()`.

```
bool operator!() const;
```

2 *Returns:* `fail()`.

```
iostate rdstate() const;
```

3 *Returns:* The error state of the stream buffer.

```
void clear(iostate state = goodbit);
```

4 *Postcondition:* If `rdbuf() != 0` then `state == rdstate()`; otherwise `rdstate() == (state | ios_base::badbit)`.

5 *Effects:* If `((state | (rdbuf() ? goodbit : badbit)) & exceptions()) == 0`, returns. Otherwise, the function throws an object of class `basic_ios::failure` (27.5.3.1.1), constructed with implementation-defined argument values.

Table 110 — `basic_ios::copyfmt()` effects

Element	Value
`rdbuf()`	*unchanged*
`tie()`	`rhs.tie()`
`rdstate()`	*unchanged*
`exceptions()`	`rhs.exceptions()`
`flags()`	`rhs.flags()`
`width()`	`rhs.width()`
`precision()`	`rhs.precision()`
`fill()`	`rhs.fill()`
`getloc()`	`rhs.getloc()`

```
void move(basic_ios& rhs);
void move(basic_ios&& rhs);
```

20 *Postconditions:* `*this` shall have the state that `rhs` had before the function call, except that `rdbuf()` shall return `nullptr`. `rhs` shall be in a valid but unspecified state, except that `rhs.rdbuf()` shall return the same value as it returned before the function call, and `rhs.tie()` shall return `nullptr`.

```
void swap(basic_ios& rhs) noexcept;
```

21 *Effects:* The states of `*this` and `rhs` shall be exchanged, except that `rdbuf()` shall return the same value as it returned before the function call, and `rhs.rdbuf()` shall return the same value as it returned before the function call.

```
void set_rdbuf(basic_streambuf<charT, traits>* sb);
```

22 *Requires:* `sb != nullptr`.

23 *Effects:* Associates the `basic_streambuf` object pointed to by `sb` with this stream without calling `clear()`.

24 *Postconditions:* `rdbuf() == sb`.

25 *Throws:* Nothing.

27.5.5.4 `basic_ios` flags functions [iostate.flags]

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

1 *Returns:* `!fail()`.

```
bool operator!() const;
```

2 *Returns:* `fail()`.

```
iostate rdstate() const;
```

3 *Returns:* The error state of the stream buffer.

```
void clear(iostate state = goodbit);
```

4 *Postcondition:* If `rdbuf() != nullptr` then `state == rdstate()`; otherwise `rdstate() == (state | ios_base::badbit)`.

5 *Effects:* If `((state | (rdbuf() ? goodbit : badbit)) & exceptions()) == 0`, returns. Otherwise, the function throws an object of class `basic_ios::failure` (27.5.3.1.1), constructed with implementation-defined argument values.

Effects: implementation-defined, except that `setbuf(0, 0)` has no effect.

Returns: `this`.

27.8.3 Class template `basic_istringstream` [istringstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
  class basic_istringstream
    : public basic_istream<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    // 27.8.3.1, constructors:
    explicit basic_istringstream(
      ios_base::openmode which = ios_base::in);
    explicit basic_istringstream(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::in);
    basic_istringstream(const basic_istringstream& rhs) = delete;
    basic_istringstream(basic_istringstream&& rhs);

    // 27.8.3.2, assign and swap:
    basic_istringstream& operator=(const basic_istringstream& rhs) = delete;
    basic_istringstream& operator=(basic_istringstream&& rhs);
    void swap(basic_istringstream& rhs);

    // 27.8.3.3, members:
    basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

    basic_string<charT, traits, Allocator> str() const;
    void str(const basic_string<charT, traits, Allocator>& s);
  private:
    basic_stringbuf<charT, traits, Allocator> sb; // exposition only
  };

  template <class charT, class traits, class Allocator>
    void swap(basic_istringstream<charT, traits, Allocator>& x,
              basic_istringstream<charT, traits, Allocator>& y);
}
```

The class `basic_istringstream<charT, traits, Allocator>` supports reading objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `stringbuf` object.

27.8.3.1 `basic_istringstream` constructors [istringstream.cons]

```
explicit basic_istringstream(ios_base::openmode which = ios_base::in);
```


Effects: implementation-defined, except that `setbuf(nullptr, 0)` has no effect.

Returns: `this`.

27.8.3 Class template `basic_istringstream` [istringstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
  class basic_istringstream
    : public basic_istream<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    // 27.8.3.1, constructors:
    explicit basic_istringstream(
      ios_base::openmode which = ios_base::in);
    explicit basic_istringstream(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::in);
    basic_istringstream(const basic_istringstream& rhs) = delete;
    basic_istringstream(basic_istringstream&& rhs);

    // 27.8.3.2, assign and swap:
    basic_istringstream& operator=(const basic_istringstream& rhs) = delete;
    basic_istringstream& operator=(basic_istringstream&& rhs);
    void swap(basic_istringstream& rhs);

    // 27.8.3.3, members:
    basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

    basic_string<charT, traits, Allocator> str() const;
    void str(const basic_string<charT, traits, Allocator>& s);
  private:
    basic_stringbuf<charT, traits, Allocator> sb; // exposition only
  };

  template <class charT, class traits, class Allocator>
    void swap(basic_istringstream<charT, traits, Allocator>& x,
              basic_istringstream<charT, traits, Allocator>& y);
}
```

The class `basic_istringstream<charT, traits, Allocator>` supports reading objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the `stringbuf` object.

27.8.3.1 `basic_istringstream` constructors [istringstream.cons]

```
explicit basic_istringstream(ios_base::openmode which = ios_base::in);
```

(10.2) — If `r == codecvt_base::noconv` then output characters from `b` up to (and not including) `p`.

(10.3) — If `r == codecvt_base::partial` then output to the file characters from `xbuf` up to `xbuf_end`, and repeat using characters from `end` to `p`. If output fails, fail (without repeating).

(10.4) — Otherwise output from `xbuf` to `xbuf_end`, and fail if output fails. At this point if `b != p` and `b == end` (`xbuf` isn't large enough) then increase `XSIZE` and repeat from the beginning.

11 *Returns:* `traits::not_eof(c)` to indicate success, and `traits::eof()` to indicate failure. If `is_open() == false`, the function always fails.

```
basic_streambuf* setbuf(char_type* s, streamsize n) override;
```

12 *Effects:* If `setbuf(0, 0)` is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined. "Unbuffered" means that `pbase()` and `pptr()` always return null and output to the file should appear as soon as possible.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out) override;
```

13 *Effects:* Let `width` denote `a_codecvt.encoding()`. If `is_open() == false`, or `off != 0 && width <= 0`, then the positioning operation fails. Otherwise, if `way != basic_ios::cur` or `off != 0`, and if the last operation was output, then update the output sequence and write any unshift sequence. Next, seek to the new position: if `width > 0`, call `std::fseek(file, width * off, whence)`, otherwise call `std::fseek(file, 0, whence)`.

14 *Remarks:* "The last operation was output" means either the last virtual operation was overflow or the put buffer is non-empty. "Write any unshift sequence" means, if `width` if less than zero then call `a_codecvt.unshift(state, xbuf, xbuf+XSIZE, xbuf_end)` and output the resulting unshift sequence. The function determines one of three values for the argument `whence`, of type `int`, as indicated in Table 114.

Table 114 — `seekoff` effects

way Value	stdio Equivalent
`basic_ios::beg`	`SEEK_SET`
`basic_ios::cur`	`SEEK_CUR`
`basic_ios::end`	`SEEK_END`

15 *Returns:* A newly constructed `pos_type` object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, returns `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out) override;
```

16 Alters the file position, if possible, to correspond to the position stored in `sp` (as described below). Altering the file position performs as follows:

1. if `(om & ios_base::out) != 0`, then update the output sequence and write any unshift sequence;
2. set the file position to `sp` as if by a call to `fsetpos`;
3. if `(om & ios_base::in) != 0`, then update the input sequence;

©ISO/IEC Dxxxx

(10.2) — If `r == codecvt_base::noconv` then output characters from `b` up to (and not including) `p`.

(10.3) — If `r == codecvt_base::partial` then output to the file characters from `xbuf` up to `xbuf_end`, and repeat using characters from `end` to `p`. If output fails, fail (without repeating).

(10.4) — Otherwise output from `xbuf` to `xbuf_end`, and fail if output fails. At this point if `b != p` and `b == end` (`xbuf` isn't large enough) then increase `XSIZE` and repeat from the beginning.

11 *Returns:* `traits::not_eof(c)` to indicate success, and `traits::eof()` to indicate failure. If `is_open() == false`, the function always fails.

```
basic_streambuf* setbuf(char_type* s, streamsize n) override;
```

12 *Effects:* If `setbuf(nullptr, 0)` is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined. "Unbuffered" means that `pbase()` and `pptr()` always return null and output to the file should appear as soon as possible.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out) override;
```

13 *Effects:* Let `width` denote `a_codecvt.encoding()`. If `is_open() == false`, or `off != 0 && width <= 0`, then the positioning operation fails. Otherwise, if `way != basic_ios::cur` or `off != 0`, and if the last operation was output, then update the output sequence and write any unshift sequence. Next, seek to the new position: if `width > 0`, call `std::fseek(file, width * off, whence)`, otherwise call `std::fseek(file, 0, whence)`.

14 *Remarks:* "The last operation was output" means either the last virtual operation was overflow or the put buffer is non-empty. "Write any unshift sequence" means, if `width` if less than zero then call `a_codecvt.unshift(state, xbuf, xbuf+XSIZE, xbuf_end)` and output the resulting unshift sequence. The function determines one of three values for the argument `whence`, of type `int`, as indicated in Table 114.

Table 114 — `seekoff` effects

way Value	stdio Equivalent
`basic_ios::beg`	`SEEK_SET`
`basic_ios::cur`	`SEEK_CUR`
`basic_ios::end`	`SEEK_END`

15 *Returns:* A newly constructed `pos_type` object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, returns `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out) override;
```

16 Alters the file position, if possible, to correspond to the position stored in `sp` (as described below). Altering the file position performs as follows:

1. if `(om & ios_base::out) != 0`, then update the output sequence and write any unshift sequence;
2. set the file position to `sp` as if by a call to `fsetpos`;
3. if `(om & ios_base::in) != 0`, then update the input sequence;

§ 27.9.2.4 1217

27.10.15.6 Create directories [fs.op.create_directories]

```
bool create_directories(const path& p);
bool create_directories(const path& p, error_code& ec) noexcept;
```

1 *Effects:* Establishes the postcondition by calling `create_directory()` for any element of `p` that does not exist.

2 *Postcondition:* `is_directory(p)`.

3 *Returns:* `true` if a new directory was created, otherwise `false`. The signature with argument `ec` returns `false` if an error occurs.

4 *Throws:* As specified in Error reporting (27.10.7).

5 *Complexity:* $\mathcal{O}(n)$ where n is the number of elements of `p` that do not exist.

27.10.15.7 Create directory [fs.op.create_directory]

```
bool create_directory(const path& p);
bool create_directory(const path& p, error_code& ec) noexcept;
```

1 *Effects:* Establishes the postcondition by attempting to create the directory `p` resolves to, as if by POSIX `mkdir()` with a second argument of `static_cast<int>(perms::all)`. Creation failure because `p` resolves to an existing directory shall not be treated as an error.

2 *Postcondition:* `is_directory(p)`.

3 *Returns:* `true` if a new directory was created, otherwise `false`. The signature with argument `ec` returns `false` if an error occurs.

4 *Throws:* As specified in Error reporting (27.10.7).

```
bool create_directory(const path& p, const path& existing_p);
bool create_directory(const path& p, const path& existing_p, error_code& ec) noexcept;
```

5 *Effects:* Establishes the postcondition by attempting to create the directory `p` resolves to, with attributes copied from directory `existing_p`. The set of attributes copied is operating system dependent. Creation failure because `p` resolves to an existing directory shall not be treated as an error. [*Note:* For POSIX-based operating systems, the attributes are those copied by native API `stat(existing_p.c_str(), &attributes_stat)` followed by `mkdir(p.c_str(), attributes_stat.st_mode)`. For Windows-based operating systems, the attributes are those copied by native API `CreateDirectoryExW(existing_p.c_str(), p.c_str(), 0)`. — *end note*]

6 *Postcondition:* `is_directory(p)`.

7 *Returns:* `true` if a new directory was created, otherwise `false`. The signature with argument `ec` returns `false` if an error occurs.

8 *Throws:* As specified in Error reporting (27.10.7).

27.10.15.8 Create directory symlink [fs.op.create_dir_symlk]

```
void create_directory_symlink(const path& to, const path& new_symlink);
void create_directory_symlink(const path& to, const path& new_symlink,
                              error_code& ec) noexcept;
```

1 *Effects:* Establishes the postcondition, as if by POSIX `symlink()`.

2 *Postcondition:* `new_symlink` resolves to a symbolic link file that contains an unspecified representation of `to`.

3 *Throws:* As specified in Error reporting (27.10.7).

27.10.15.6 Create directories [fs.op.create_directories]

```
bool create_directories(const path& p);
bool create_directories(const path& p, error_code& ec) noexcept;
```

1 *Effects:* Establishes the postcondition by calling `create_directory()` for any element of `p` that does not exist.

2 *Postcondition:* `is_directory(p)`.

3 *Returns:* `true` if a new directory was created, otherwise `false`. The signature with argument `ec` returns `false` if an error occurs.

4 *Throws:* As specified in Error reporting (27.10.7).

5 *Complexity:* $\mathcal{O}(n)$ where n is the number of elements of `p` that do not exist.

27.10.15.7 Create directory [fs.op.create_directory]

```
bool create_directory(const path& p);
bool create_directory(const path& p, error_code& ec) noexcept;
```

1 *Effects:* Establishes the postcondition by attempting to create the directory `p` resolves to, as if by POSIX `mkdir()` with a second argument of `static_cast<int>(perms::all)`. Creation failure because `p` resolves to an existing directory shall not be treated as an error.

2 *Postcondition:* `is_directory(p)`.

3 *Returns:* `true` if a new directory was created, otherwise `false`. The signature with argument `ec` returns `false` if an error occurs.

4 *Throws:* As specified in Error reporting (27.10.7).

```
bool create_directory(const path& p, const path& existing_p);
bool create_directory(const path& p, const path& existing_p, error_code& ec) noexcept;
```

5 *Effects:* Establishes the postcondition by attempting to create the directory `p` resolves to, with attributes copied from directory `existing_p`. The set of attributes copied is operating system dependent. Creation failure because `p` resolves to an existing directory shall not be treated as an error. [*Note:* For POSIX-based operating systems, the attributes are those copied by native API `stat(existing_p.c_str(), &attributes_stat)` followed by `mkdir(p.c_str(), attributes_stat.st_mode)`. For Windows-based operating systems, the attributes are those copied by native API `CreateDirectoryExW(existing_p.c_str(), p.c_str(), nullptr)`. — *end note*]

6 *Postcondition:* `is_directory(p)`.

7 *Returns:* `true` if a new directory was created, otherwise `false`. The signature with argument `ec` returns `false` if an error occurs.

8 *Throws:* As specified in Error reporting (27.10.7).

27.10.15.8 Create directory symlink [fs.op.create_dir_symlk]

```
void create_directory_symlink(const path& to, const path& new_symlink);
void create_directory_symlink(const path& to, const path& new_symlink,
                              error_code& ec) noexcept;
```

1 *Effects:* Establishes the postcondition, as if by POSIX `symlink()`.

2 *Postcondition:* `new_symlink` resolves to a symbolic link file that contains an unspecified representation of `to`.

3 *Throws:* As specified in Error reporting (27.10.7).

§ 27.10.15.8 1265

```
  private:
    mutex_type* pm; // exposition only
    bool owns;      // exposition only
  };

  template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
}
```

1 An object of type `unique_lock` controls the ownership of a lockable object within a scope. Ownership of the lockable object may be acquired at construction or after construction, and may be transferred, after acquisition, to another `unique_lock` object. Objects of type `unique_lock` are not copyable but are movable. The behavior of a program is undefined if the contained pointer `pm` is not null and the lockable object pointed to by `pm` does not exist for the entire remaining lifetime (3.8) of the `unique_lock` object. The supplied `Mutex` type shall meet the `BasicLockable` requirements (30.2.5.2).

2 [*Note:* `unique_lock<Mutex>` meets the `BasicLockable` requirements. If `Mutex` meets the `Lockable` requirements (30.2.5.3), `unique_lock<Mutex>` also meets the `Lockable` requirements; if `Mutex` meets the `TimedLockable` requirements (30.2.5.4), `unique_lock<Mutex>` also meets the `TimedLockable` requirements. — *end note*]

30.4.2.2.1 `unique_lock` constructors, destructor, and assignment [thread.lock.unique.cons]

```
unique_lock() noexcept;
```

1 *Effects:* Constructs an object of type `unique_lock`.

2 *Postconditions:* `pm == 0` and `owns == false`.

```
explicit unique_lock(mutex_type& m);
```

3 *Requires:* If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

4 *Effects:* Constructs an object of type `unique_lock` and calls `m.lock()`.

5 *Postconditions:* `pm == addressof(m)` and `owns == true`.

```
unique_lock(mutex_type& m, defer_lock_t) noexcept;
```

6 *Effects:* Constructs an object of type `unique_lock`.

7 *Postconditions:* `pm == addressof(m)` and `owns == false`.

```
unique_lock(mutex_type& m, try_to_lock_t);
```

8 *Requires:* The supplied `Mutex` type shall meet the `Lockable` requirements (30.2.5.3). If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

9 *Effects:* Constructs an object of type `unique_lock` and calls `m.try_lock()`.

10 *Postconditions:* `pm == addressof(m)` and `owns == res`, where `res` is the value returned by the call to `m.try_lock()`.

```
unique_lock(mutex_type& m, adopt_lock_t);
```

11 *Requires:* The calling thread owns the mutex.

12 *Effects:* Constructs an object of type `unique_lock`.

13 *Postconditions:* `pm == addressof(m)` and `owns == true`.

14 *Throws:* Nothing.


```
  private:
    mutex_type* pm; // exposition only
    bool owns;      // exposition only
  };

  template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
}
```

1 An object of type `unique_lock` controls the ownership of a lockable object within a scope. Ownership of the lockable object may be acquired at construction or after construction, and may be transferred, after acquisition, to another `unique_lock` object. Objects of type `unique_lock` are not copyable but are movable. The behavior of a program is undefined if the contained pointer `pm` is not null and the lockable object pointed to by `pm` does not exist for the entire remaining lifetime (3.8) of the `unique_lock` object. The supplied `Mutex` type shall meet the `BasicLockable` requirements (30.2.5.2).

2 [*Note:* `unique_lock<Mutex>` meets the `BasicLockable` requirements. If `Mutex` meets the `Lockable` requirements (30.2.5.3), `unique_lock<Mutex>` also meets the `Lockable` requirements; if `Mutex` meets the `TimedLockable` requirements (30.2.5.4), `unique_lock<Mutex>` also meets the `TimedLockable` requirements. — *end note*]

30.4.2.2.1 `unique_lock` constructors, destructor, and assignment [thread.lock.unique.cons]

```
unique_lock() noexcept;
```

1 *Effects:* Constructs an object of type `unique_lock`.

2 *Postconditions:* `pm == nullptr` and `owns == false`.

```
explicit unique_lock(mutex_type& m);
```

3 *Requires:* If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

4 *Effects:* Constructs an object of type `unique_lock` and calls `m.lock()`.

5 *Postconditions:* `pm == addressof(m)` and `owns == true`.

```
unique_lock(mutex_type& m, defer_lock_t) noexcept;
```

6 *Effects:* Constructs an object of type `unique_lock`.

7 *Postconditions:* `pm == addressof(m)` and `owns == false`.

```
unique_lock(mutex_type& m, try_to_lock_t);
```

8 *Requires:* The supplied `Mutex` type shall meet the `Lockable` requirements (30.2.5.3). If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

9 *Effects:* Constructs an object of type `unique_lock` and calls `m.try_lock()`.

10 *Postconditions:* `pm == addressof(m)` and `owns == res`, where `res` is the value returned by the call to `m.try_lock()`.

```
unique_lock(mutex_type& m, adopt_lock_t);
```

11 *Requires:* The calling thread owns the mutex.

12 *Effects:* Constructs an object of type `unique_lock`.

13 *Postconditions:* `pm == addressof(m)` and `owns == true`.

14 *Throws:* Nothing.

```
template <class Clock, class Duration>
  unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
```

Requires: If `mutex_type` is not a recursive mutex the calling thread does not own the mutex. The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.2.5.4).

Effects: Constructs an object of type `unique_lock` and calls `m.try_lock_until(abs_time)`.

Postconditions: `pm == addressof(m)` and `owns == res`, where `res` is the value returned by the call to `m.try_lock_until(abs_time)`.

```
template <class Rep, class Period>
  unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
```

Requires: If `mutex_type` is not a recursive mutex the calling thread does not own the mutex. The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.2.5.4).

Effects: Constructs an object of type `unique_lock` and calls `m.try_lock_for(rel_time)`.

Postconditions: `pm == addressof(m)` and `owns == res`, where `res` is the value returned by the call to `m.try_lock_for(rel_time)`.

```
unique_lock(unique_lock&& u) noexcept;
```

Postconditions: `pm == u_p.pm` and `owns == u_p.owns` (where `u_p` is the state of `u` just prior to this construction), `u.pm == 0` and `u.owns == false`.

```
unique_lock& operator=(unique_lock&& u);
```

Effects: If `owns` calls `pm->unlock()`.

Postconditions: `pm == u_p.pm` and `owns == u_p.owns` (where `u_p` is the state of `u` just prior to this construction), `u.pm == 0` and `u.owns == false`.

[*Note:* With a recursive mutex it is possible for both `*this` and `u` to own the same mutex before the assignment. In this case, `*this` will own the mutex after the assignment and `u` will not. — *end note*]

Throws: Nothing.

```
~unique_lock();
```

Effects: If `owns` calls `pm->unlock()`.

30.4.2.2.2 `unique_lock` locking [thread.lock.unique.locking]

```
void lock();
```

Effects: As if by `pm->lock()`.

Postcondition: `owns == true`

Throws: Any exception thrown by `pm->lock()`. `system_error` if an exception is required (30.2.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is `0`. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
bool try_lock();
```

Requires: The supplied `Mutex` shall meet the `Lockable` requirements (30.2.5.3).

Effects: As if by `pm->try_lock()`.

Returns: The value returned by the call to `try_lock()`.

Postcondition: `owns == res`, where `res` is the value returned by the call to `try_lock()`.

Throws: Any exception thrown by `pm->try_lock()`. `system_error` if an exception is required (30.2.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is `0`. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.


```
template <class Clock, class Duration>
  unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
```

Requires: If `mutex_type` is not a recursive mutex the calling thread does not own the mutex. The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.2.5.4).

Effects: Constructs an object of type `unique_lock` and calls `m.try_lock_until(abs_time)`.

Postconditions: `pm == addressof(m)` and `owns == res`, where `res` is the value returned by the call to `m.try_lock_until(abs_time)`.

```
template <class Rep, class Period>
  unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
```

Requires: If `mutex_type` is not a recursive mutex the calling thread does not own the mutex. The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.2.5.4).

Effects: Constructs an object of type `unique_lock` and calls `m.try_lock_for(rel_time)`.

Postconditions: `pm == addressof(m)` and `owns == res`, where `res` is the value returned by the call to `m.try_lock_for(rel_time)`.

```
unique_lock(unique_lock&& u) noexcept;
```

Postconditions: `pm == u_p.pm` and `owns == u_p.owns` (where `u_p` is the state of `u` just prior to this construction), `u.pm == nullptr` and `u.owns == false`.

```
unique_lock& operator=(unique_lock&& u);
```

Effects: If `owns` calls `pm->unlock()`.

Postconditions: `pm == u_p.pm` and `owns == u_p.owns` (where `u_p` is the state of `u` just prior to this construction), `u.pm == nullptr` and `u.owns == false`.

[*Note:* With a recursive mutex it is possible for both `*this` and `u` to own the same mutex before the assignment. In this case, `*this` will own the mutex after the assignment and `u` will not. — *end note*]

Throws: Nothing.

```
~unique_lock();
```

Effects: If `owns` calls `pm->unlock()`.

30.4.2.2.2 `unique_lock` locking [thread.lock.unique.locking]

```
void lock();
```

Effects: As if by `pm->lock()`.

Postcondition: `owns == true`

Throws: Any exception thrown by `pm->lock()`. `system_error` if an exception is required (30.2.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is `nullptr`. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
bool try_lock();
```

Requires: The supplied `Mutex` shall meet the `Lockable` requirements (30.2.5.3).

Effects: As if by `pm->try_lock()`.

Returns: The value returned by the call to `try_lock()`.

Postcondition: `owns == res`, where `res` is the value returned by the call to `try_lock()`.

Throws: Any exception thrown by `pm->try_lock()`. `system_error` if an exception is required (30.2.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is `nullptr`. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

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

Requires: The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.2.5.4).

Effects: As if by `pm->try_lock_until(abs_time)`.

Returns: The value returned by the call to `try_lock_until(abs_time)`.

Postcondition: `owns == res`, where `res` is the value returned by the call to `try_lock_until(abs_time)`.

Throws: Any exception thrown by `pm->try_lock_until()`. `system_error` if an exception is required (30.2.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is 0. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

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

Requires: The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.2.5.4).

Effects: As if by `pm->try_lock_for(rel_time)`.

Returns: The value returned by the call to `try_lock_until(rel_time)`.

Postcondition: `owns == res`, where `res` is the value returned by the call to `try_lock_for(rel_time)`.

Throws: Any exception thrown by `pm->try_lock_for()`. `system_error` if an exception is required (30.2.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is 0. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
void unlock();
```

Effects: As if by `pm->unlock()`.

Postcondition: `owns == false`

Throws: `system_error` when an exception is required (30.2.2).

Error conditions:

(22.1) — `operation_not_permitted` — if on entry `owns` is false.

30.4.2.2.3 `unique_lock` modifiers [thread.lock.unique.mod]

```
void swap(unique_lock& u) noexcept;
```

Effects: Swaps the data members of `*this` and `u`.

```
mutex_type* release() noexcept;
```

Returns: The previous value of `pm`.

Postconditions: `pm == 0` and `owns == false`.

```
template <class Mutex>
  void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
```

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


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

Requires: The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.2.5.4).

Effects: As if by `pm->try_lock_until(abs_time)`.

Returns: The value returned by the call to `try_lock_until(abs_time)`.

Postcondition: `owns == res`, where `res` is the value returned by the call to `try_lock_until(abs_time)`.

Throws: Any exception thrown by `pm->try_lock_until()`. `system_error` if an exception is required (30.2.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is `nullptr`. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

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

Requires: The supplied `Mutex` type shall meet the `TimedLockable` requirements (30.2.5.4).

Effects: As if by `pm->try_lock_for(rel_time)`.

Returns: The value returned by the call to `try_lock_until(rel_time)`.

Postcondition: `owns == res`, where `res` is the value returned by the call to `try_lock_for(rel_time)`.

Throws: Any exception thrown by `pm->try_lock_for()`. `system_error` if an exception is required (30.2.2). `system_error` with an error condition of `operation_not_permitted` if `pm` is `nullptr`. `system_error` with an error condition of `resource_deadlock_would_occur` if on entry `owns` is `true`.

```
void unlock();
```

Effects: As if by `pm->unlock()`.

Postcondition: `owns == false`

Throws: `system_error` when an exception is required (30.2.2).

Error conditions:

(22.1) — `operation_not_permitted` — if on entry `owns` is false.

30.4.2.2.3 `unique_lock` modifiers [thread.lock.unique.mod]

```
void swap(unique_lock& u) noexcept;
```

Effects: Swaps the data members of `*this` and `u`.

```
mutex_type* release() noexcept;
```

Returns: The previous value of `pm`.

Postconditions: `pm == nullptr` and `owns == false`.

```
template <class Mutex>
  void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
```

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

placed within the global namespace scope. It is unspecified whether these names are first declared or defined within namespace scope (3.3.6) of the namespace `std` and are then injected into the global namespace scope by explicit *using-declarations* (7.3.3).

4 [*Example:* The header `<cstdlib>` assuredly provides its declarations and definitions within the namespace `std`. It may also provide these names within the global namespace. The header `<stdlib.h>` assuredly provides the same declarations and definitions within the global namespace, much as in the C Standard. It may also provide these names within the namespace `std`. — *end example*]

D.5 char* streams [depr.str.strstreams]

1 The header `<strstream>` defines three types that associate stream buffers with character array objects and assist reading and writing such objects.

D.5.1 Class strstreambuf [depr.strstreambuf]

```
namespace std {
  class strstreambuf : public basic_streambuf<char> {
  public:
    explicit strstreambuf(streamsize alsize_arg = 0);
    strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
    strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
    strstreambuf(const char* gnext_arg, streamsize n);

    strstreambuf(signed char* gnext_arg, streamsize n,
                 signed char* pbeg_arg = 0);
    strstreambuf(const signed char* gnext_arg, streamsize n);
    strstreambuf(unsigned char* gnext_arg, streamsize n,
                 unsigned char* pbeg_arg = 0);
    strstreambuf(const unsigned char* gnext_arg, streamsize n);

    virtual ~strstreambuf();

    void  freeze(bool freezefl = true);
    char* str();
    int   pcount();

  protected:
    int_type overflow (int_type c = EOF) override;
    int_type pbackfail(int_type c = EOF) override;
    int_type underflow() override;
    pos_type seekoff(off_type off, ios_base::seekdir way,
                     ios_base::openmode which
                      = ios_base::in | ios_base::out) override;
    pos_type seekpos(pos_type sp,
                     ios_base::openmode which
                      = ios_base::in | ios_base::out) override;
    streambuf* setbuf(char* s, streamsize n) override;

  private:
    using strstate = T1;              // exposition only
    static const strstate allocated;  // exposition only
    static const strstate constant;   // exposition only
    static const strstate dynamic;    // exposition only
    static const strstate frozen;     // exposition only
    strstate strmode;                 // exposition only
```


©ISO/IEC Dxxxx

placed within the global namespace scope. It is unspecified whether these names are first declared or defined within namespace scope (3.3.6) of the namespace `std` and are then injected into the global namespace scope by explicit *using-declarations* (7.3.3).

4 [*Example:* The header `<cstdlib>` assuredly provides its declarations and definitions within the namespace `std`. It may also provide these names within the global namespace. The header `<stdlib.h>` assuredly provides the same declarations and definitions within the global namespace, much as in the C Standard. It may also provide these names within the namespace `std`. — *end example*]

D.5 char* streams [depr.str.strstreams]

1 The header `<strstream>` defines three types that associate stream buffers with character array objects and assist reading and writing such objects.

D.5.1 Class strstreambuf [depr.strstreambuf]

```
namespace std {
  class strstreambuf : public basic_streambuf<char> {
  public:
    explicit strstreambuf(streamsize alsize_arg = 0);
    strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
    strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = nullptr);
    strstreambuf(const char* gnext_arg, streamsize n);

    strstreambuf(signed char* gnext_arg, streamsize n,
                 signed char* pbeg_arg = nullptr);
    strstreambuf(const signed char* gnext_arg, streamsize n);
    strstreambuf(unsigned char* gnext_arg, streamsize n,
                 unsigned char* pbeg_arg = nullptr);
    strstreambuf(const unsigned char* gnext_arg, streamsize n);

    virtual ~strstreambuf();

    void  freeze(bool freezefl = true);
    char* str();
    int   pcount();

  protected:
    int_type overflow (int_type c = EOF) override;
    int_type pbackfail(int_type c = EOF) override;
    int_type underflow() override;
    pos_type seekoff(off_type off, ios_base::seekdir way,
                     ios_base::openmode which
                      = ios_base::in | ios_base::out) override;
    pos_type seekpos(pos_type sp,
                     ios_base::openmode which
                      = ios_base::in | ios_base::out) override;
    streambuf* setbuf(char* s, streamsize n) override;

  private:
    using strstate = T1;              // exposition only
    static const strstate allocated;  // exposition only
    static const strstate constant;   // exposition only
    static const strstate dynamic;    // exposition only
    static const strstate frozen;     // exposition only
    strstate strmode;                 // exposition only
```

Table 145 — `strstreambuf(void* (*)(size_t), void (*)(void*))` effects

Element	Value
strmode	dynamic
alsize	an unspecified value
palloc	palloc_arg
pfree	pfree_arg

2 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 145.

```
strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
strstreambuf(signed char* gnext_arg, streamsize n,
             signed char* pbeg_arg = 0);
strstreambuf(unsigned char* gnext_arg, streamsize n,
             unsigned char* pbeg_arg = 0);
```

3 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 146.

Table 146 — `strstreambuf(charT*, streamsize, charT*)` effects

Element	Value
strmode	0
alsize	an unspecified value
palloc	a null pointer
pfree	a null pointer

4 `gnext_arg` shall point to the first element of an array object whose number of elements `N` is determined as follows:

(4.1) — If `n > 0`, `N` is `n`.

(4.2) — If `n == 0`, `N` is `std::strlen(gnext_arg)`.

(4.3) — If `n < 0`, `N` is `INT_MAX`.[336]

5 If `pbeg_arg` is a null pointer, the function executes:

```
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

6 Otherwise, the function executes:

```
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg,  pbeg_arg + N);

strstreambuf(const char* gnext_arg, streamsize n);
strstreambuf(const signed char* gnext_arg, streamsize n);
strstreambuf(const unsigned char* gnext_arg, streamsize n);
```

7 *Effects:* Behaves the same as `strstreambuf((char*)gnext_arg,n)`, except that the constructor also sets `constant` in `strmode`.

```
virtual ~strstreambuf();
```

336) The function signature `strlen(const char*)` is declared in `<cstring>`. (21.5). The macro `INT_MAX` is defined in `<climits>` (18.3).

Table 145 — `strstreambuf(void* (*)(size_t), void (*)(void*))` effects

Element	Value
strmode	dynamic
alsize	an unspecified value
palloc	palloc_arg
pfree	pfree_arg

2 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 145.

```
strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = nullptr);
strstreambuf(signed char* gnext_arg, streamsize n,
             signed char* pbeg_arg = nullptr);
strstreambuf(unsigned char* gnext_arg, streamsize n,
             unsigned char* pbeg_arg = nullptr);
```

3 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 146.

Table 146 — `strstreambuf(charT*, streamsize, charT*)` effects

Element	Value
strmode	0
alsize	an unspecified value
palloc	a null pointer
pfree	a null pointer

4 `gnext_arg` shall point to the first element of an array object whose number of elements `N` is determined as follows:

(4.1) — If `n > 0`, `N` is `n`.

(4.2) — If `n == 0`, `N` is `std::strlen(gnext_arg)`.

(4.3) — If `n < 0`, `N` is `INT_MAX`.[336]

5 If `pbeg_arg` is a null pointer, the function executes:

```
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

6 Otherwise, the function executes:

```
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg,  pbeg_arg + N);

strstreambuf(const char* gnext_arg, streamsize n);
strstreambuf(const signed char* gnext_arg, streamsize n);
strstreambuf(const unsigned char* gnext_arg, streamsize n);
```

7 *Effects:* Behaves the same as `strstreambuf((char*)gnext_arg,n)`, except that the constructor also sets `constant` in `strmode`.

```
virtual ~strstreambuf();
```

336) The function signature `strlen(const char*)` is declared in `<cstring>`. (21.5). The macro `INT_MAX` is defined in `<climits>` (18.3).

Table 148 — `newoff` values

Condition	`newoff` Value
`way == ios::beg`	0
`way == ios::cur`	the next pointer minus the beginning pointer (`xnext - xbeg`).
`way == ios::end`	`seekhigh` minus the beginning pointer (`seekhigh - xbeg`).

15 If `(newoff + off) < (seeklow - xbeg)` or `(seekhigh - xbeg) < (newoff + off)`, the positioning operation fails. Otherwise, the function assigns `xbeg + newoff + off` to the next pointer `xnext`.

16 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp, ios_base::openmode which
                 = ios_base::in | ios_base::out) override;
```

17 *Effects:* Alters the stream position within one of the controlled sequences, if possible, to correspond to the stream position stored in `sp` (as described below).

(17.1) — If `(which & ios::in) != 0`, positions the input sequence.

(17.2) — If `(which & ios::out) != 0`, positions the output sequence.

(17.3) — If the function positions neither sequence, the positioning operation fails.

18 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines `newoff` from `sp.offset()`:

(18.1) — If `newoff` is an invalid stream position, has a negative value, or has a value greater than `(seekhigh - seeklow)`, the positioning operation fails

(18.2) — Otherwise, the function adds `newoff` to the beginning pointer `xbeg` and stores the result in the next pointer `xnext`.

19 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
streambuf<char>* setbuf(char* s, streamsize n) override;
```

20 *Effects:* Implementation defined, except that `setbuf(0, 0)` has no effect.

D.5.2 Class `istrstream` [depr.istrstream]

```
namespace std {
  class istrstream : public basic_istream<char> {
  public:
    explicit istrstream(const char* s);
    explicit istrstream(char* s);
    istrstream(const char* s, streamsize n);
    istrstream(char* s, streamsize n);
    virtual ~istrstream();

    strstreambuf* rdbuf() const;
    char* str();
```


Table 148 — `newoff` values

Condition	`newoff` Value
`way == ios::beg`	0
`way == ios::cur`	the next pointer minus the beginning pointer (`xnext - xbeg`).
`way == ios::end`	`seekhigh` minus the beginning pointer (`seekhigh - xbeg`).

15 If `(newoff + off) < (seeklow - xbeg)` or `(seekhigh - xbeg) < (newoff + off)`, the positioning operation fails. Otherwise, the function assigns `xbeg + newoff + off` to the next pointer `xnext`.

16 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp, ios_base::openmode which
                 = ios_base::in | ios_base::out) override;
```

17 *Effects:* Alters the stream position within one of the controlled sequences, if possible, to correspond to the stream position stored in `sp` (as described below).

(17.1) — If `(which & ios::in) != 0`, positions the input sequence.

(17.2) — If `(which & ios::out) != 0`, positions the output sequence.

(17.3) — If the function positions neither sequence, the positioning operation fails.

18 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines `newoff` from `sp.offset()`:

(18.1) — If `newoff` is an invalid stream position, has a negative value, or has a value greater than `(seekhigh - seeklow)`, the positioning operation fails

(18.2) — Otherwise, the function adds `newoff` to the beginning pointer `xbeg` and stores the result in the next pointer `xnext`.

19 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
streambuf<char>* setbuf(char* s, streamsize n) override;
```

20 *Effects:* Implementation defined, except that `setbuf(nullptr, 0)` has no effect.

D.5.2 Class `istrstream` [depr.istrstream]

```
namespace std {
  class istrstream : public basic_istream<char> {
  public:
    explicit istrstream(const char* s);
    explicit istrstream(char* s);
    istrstream(const char* s, streamsize n);
    istrstream(char* s, streamsize n);
    virtual ~istrstream();

    strstreambuf* rdbuf() const;
    char* str();
```

```
template <class U>
  void destroy(U* p);
```

7 *Effects:* As if by `p->~U()`.

```
size_t max_size() const noexcept;
```

8 *Returns:* The largest value N for which the call `allocate(N, 0)` might succeed.

D.10 Raw storage iterator [depr.storage.iterator]

1 The header `<memory>` has the following addition:

```
namespace std {
  template <class OutputIterator, class T>
  class raw_storage_iterator {
  public:
    using iterator_category = output_iterator_tag;
    using value_type        = void;
    using difference_type   = void;
    using pointer           = void;
    using reference         = void;

    explicit raw_storage_iterator(OutputIterator x);

    raw_storage_iterator& operator*();
    raw_storage_iterator& operator=(const T& element);
    raw_storage_iterator& operator=(T&& element);
    raw_storage_iterator& operator++();
    raw_storage_iterator  operator++(int);
    OutputIterator base() const;
  };
}
```

2 `raw_storage_iterator` is provided to enable algorithms to store their results into uninitialized memory. The template parameter `OutputIterator` is required to have its `operator*` return an object for which `operator&` is defined and returns a pointer to `T`, and is also required to satisfy the requirements of an output iterator (24.2.4).

```
explicit raw_storage_iterator(OutputIterator x);
```

3 *Effects:* Initializes the iterator to point to the same value to which `x` points.

```
raw_storage_iterator& operator*();
```

4 *Returns:* `*this`

```
raw_storage_iterator& operator=(const T& element);
```

5 *Requires:* `T` shall be `CopyConstructible`.

6 *Effects:* Constructs a value from `element` at the location to which the iterator points.

7 *Returns:* A reference to the iterator.

```
raw_storage_iterator& operator=(T&& element);
```

8 *Requires:* `T` shall be `MoveConstructible`.

9 *Effects:* Constructs a value from `std::move(element)` at the location to which the iterator points.

10 *Returns:* A reference to the iterator.

©ISO/IEC Dxxxx

```
template <class U>
  void destroy(U* p);
```

7 *Effects:* As if by `p->~U()`.

```
size_t max_size() const noexcept;
```

8 *Returns:* The largest value N for which the call `allocate(N, nullptr)` might succeed.

D.10 Raw storage iterator [depr.storage.iterator]

1 The header `<memory>` has the following addition:

```
namespace std {
  template <class OutputIterator, class T>
  class raw_storage_iterator {
  public:
    using iterator_category = output_iterator_tag;
    using value_type        = void;
    using difference_type   = void;
    using pointer           = void;
    using reference         = void;

    explicit raw_storage_iterator(OutputIterator x);

    raw_storage_iterator& operator*();
    raw_storage_iterator& operator=(const T& element);
    raw_storage_iterator& operator=(T&& element);
    raw_storage_iterator& operator++();
    raw_storage_iterator  operator++(int);
    OutputIterator base() const;
  };
}
```

2 `raw_storage_iterator` is provided to enable algorithms to store their results into uninitialized memory. The template parameter `OutputIterator` is required to have its `operator*` return an object for which `operator&` is defined and returns a pointer to `T`, and is also required to satisfy the requirements of an output iterator (24.2.4).

```
explicit raw_storage_iterator(OutputIterator x);
```

3 *Effects:* Initializes the iterator to point to the same value to which `x` points.

```
raw_storage_iterator& operator*();
```

4 *Returns:* `*this`

```
raw_storage_iterator& operator=(const T& element);
```

5 *Requires:* `T` shall be `CopyConstructible`.

6 *Effects:* Constructs a value from `element` at the location to which the iterator points.

7 *Returns:* A reference to the iterator.

```
raw_storage_iterator& operator=(T&& element);
```

8 *Requires:* `T` shall be `MoveConstructible`.

9 *Effects:* Constructs a value from `std::move(element)` at the location to which the iterator points.

10 *Returns:* A reference to the iterator.

§ D.10 1462