Document Number: P1719R2
Date: 2019-11-08
Reply to: Marshall Clow
mclow.lists@gmail.com

Mandating the Standard Library: Clause 26 - Numerics library

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

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

- Change "participate in overload resolution" wording into "Constraints" elements
- Change "Requires" elements into either "Mandates" or "Expects", depending (mostly) on whether or not they can be checked at compile time.
- Drive-by fixes (hopefully very few)

This paper covers Clause 26 (Numerics), and is based on N4830.

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

- cfenv.syn 26.3.1
- complex.numbers 26.4
- complex.members 26.4.4
- complex.ops 26.4.6
- complex.value.ops 26.4.7
- bit.cast 26.5.3
- bit.pow.two 26.5.4
- rand.req.eng 26.6.2.4
- rand.req.dist 26.6.2.6
- rand.eng.lcong 26.6.3.1
- rand.eng.mers 26.6.3.2
- rand.eng.sub 26.6.3.3
- rand.device 26.6.6
- rand.util.seedseq 26.6.7.1
- rand.dist.uni.int 26.6.8.2.1
- rand.dist.uni.real 26.6.8.2.2
- rand.dist.bern.bernoulli 26.6.8.3.1
- rand.dist.bern.bin 26.6.8.3.2
- rand.dist.bern.geo 26.6.8.3.3
- rand.dist.bern.negbin 26.6.8.3.4
- rand.dist.pois.poisson 26.6.8.4.1
- rand.dist.pois.exp 26.6.8.4.2
- rand.dist.pois.gamma 26.6.8.4.3
- rand.dist.pois.weibull 26.6.8.4.4
- rand.dist.pois.extreme 26.6.8.4.5
- rand.dist.norm.normal 26.6.8.5.1
- rand.dist.norm.lognormal 26.6.8.5.2
- rand.dist.norm.chisq 26.6.8.5.3
- rand.dist.norm.cauchy 26.6.8.5.4
- rand.dist.norm.f 26.6.8.5.5
- rand.dist.norm.t 26.6.8.5.6
- rand.dist.samp.discrete 26.6.8.6.1
- rand.dist.samp.pconst 26.6.8.6.2
- rand.dist.samp.plinear 26.6.8.6.3
- valarray.cons 26.7.2.2
- valarray.assign 26.7.2.3
- valarray.access 26.7.2.4
- valarray.unary 26.7.2.6
- valarray.cassign 26.7.2.7
- valarray.members 26.7.2.8
- valarray.binary 26.7.3.1
- valarray.comparison 26.7.3.2
- valarray.transcend 26.7.3.3

Drive-by fixes:

- Removed several useless "Constructs an object of type XXXX" sentences, and reworked a bunch left-over "Effects" into "Remarks"
- While moving "Expects" to "Mandates", changed "XXXX shall denote a type that is convertible to double" to "`is_convertible_v<XXXX, double>` is true."

- — While moving "Expects" to "Mandates", changed "XXXX shall be callable with a type that is convertible to double" to "`is_invocable_r_v<XXXX, double>` is true."
- — Changed a "points to an array of at least n elements" -> "`[p, p+n)` is a valid range."
- — Changed several "Complexity" clauses from "shall not exceed" to "does not exceed".
- — Reworked two paragraphs in [rand.util.seedseq] to use the exposition-only variable `v`.

Open questions:

- — The "xxx relation holds" formulation needs a pattern.

Changes from R0:

- — Rebased on N4830
- — Removed several "shall"s from [cfenv.syn] (26.3.1) and [complex.numbers] (26.4)
- — Changed a "points to an array of at least N" -> "`[p, p+n) is a valid range`"
- — Turned a couple of sentences into bullet points in [rand.dist.samp.pconst] and prand.dist.samp.plinear].
- — Items discussed during the LWG telecom on 6-Sep-2019.
- — Reworked two paragraphs in [rand.util.seedseq] to use the exposition-only variable `v`.

Changes from R1:

- — Changes from LWG review in Belfast

Thanks to Daniel Krügler for his several rounds of review.

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

```
git clone git@github.com:mclow/mandate.git
cd mandate
git diff master..chapter26 numerics.tex
```

26 Numerics library [numerics]

26.1 General [numerics.general]

1 This Clause describes components that C++ programs may use to perform seminumerical operations.

2 The following subclauses describe components for complex number types, random number generation, numeric (n-at-a-time) arrays, generalized numeric algorithms, and mathematical constants and functions for floating-point types, as summarized in Table 91.

Table 91: Numerics library summary [tab:numerics.summary]

	Subclause	Header
26.2	Requirements	
26.3	Floating-point environment	`<cfenv>`
26.4	Complex numbers	`<complex>`
26.5	Bit manipulation	`<bit>`
26.6	Random number generation	`<random>`
26.7	Numeric arrays	`<valarray>`
26.8	Mathematical functions for floating-point types	`<cmath>`, `<cstdlib>`
26.9	Numbers	`<numbers>`

26.2 Numeric type requirements [numeric.requirements]

1 The `complex` and `valarray` components are parameterized by the type of information they contain and manipulate. A C++ program shall instantiate these components only with a numeric type. A *numeric type* is a cv-unqualified object type `T` that meets the *Cpp17DefaultConstructible*, *Cpp17CopyConstructible*, *Cpp17CopyAssignable*, and *Cpp17Destructible* requirements (**??**).[241]

2 If any operation on `T` throws an exception the effects are undefined.

3 In addition, many member and related functions of `valarray<T>` can be successfully instantiated and will exhibit well-defined behavior if and only if `T` meets additional requirements specified for each such member or related function.

4 [*Example*: It is valid to instantiate `valarray<complex>`, but `operator>()` will not be successfully instantiated for `valarray<complex>` operands, since `complex` does not have any ordering operators. — *end example*]

26.3 The floating-point environment [cfenv]

26.3.1 Header `<cfenv>` synopsis [cfenv.syn]

```
#define FE_ALL_EXCEPT see below
#define FE_DIVBYZERO see below      // optional
#define FE_INEXACT see below        // optional
#define FE_INVALID see below        // optional
#define FE_OVERFLOW see below       // optional
#define FE_UNDERFLOW see below      // optional

#define FE_DOWNWARD see below       // optional
#define FE_TONEAREST see below      // optional
#define FE_TOWARDZERO see below     // optional
#define FE_UPWARD see below         // optional

#define FE_DFL_ENV see below
```

241) In other words, value types. These include arithmetic types, pointers, the library class `complex`, and instantiations of `valarray` for value types.

```
namespace std {
  // types
  using fenv_t    = object type;
  using fexcept_t = integer type;

  // functions
  int feclearexcept(int except);
  int fegetexceptflag(fexcept_t* pflag, int except);
  int feraiseexcept(int except);
  int fesetexceptflag(const fexcept_t* pflag, int except);
  int fetestexcept(int except);

  int fegetround();
  int fesetround(int mode);

  int fegetenv(fenv_t* penv);
  int feholdexcept(fenv_t* penv);
  int fesetenv(const fenv_t* penv);
  int feupdateenv(const fenv_t* penv);
}
```

1 The contents and meaning of the header `<cfenv>` are the same as the C standard library header `<fenv.h>`. [*Note*: This document does not require an implementation to support the `FENV_ACCESS` pragma; it is implementation-defined (??) whether the pragma is supported. As a consequence, it is implementation-defined whether these functions can be used to test floating-point status flags, set floating-point control modes, or run under non-default mode settings. If the pragma is used to enable control over the floating-point environment, this document does not specify the effect on floating-point evaluation in constant expressions. — *end note*]

2 The floating-point environment has thread storage duration (??). The initial state for a thread's floating-point environment is the state of the floating-point environment of the thread that constructs the corresponding `thread` object (??) or `jthread` object (??) at the time it constructed the object. [*Note*: That is, the child thread gets the floating-point state of the parent thread at the time of the child's creation. — *end note*]

3 A separate floating-point environment ~~shall be~~is maintained for each thread. Each function accesses the environment corresponding to its calling thread.

SEE ALSO: ISO C 7.6

26.4 Complex numbers [complex.numbers]

1 The header `<complex>` defines a class template, and numerous functions for representing and manipulating complex numbers.

2 The effect of instantiating the template `complex` for any type other than `float`, `double`, or `long double` is unspecified. The specializations `complex<float>`, `complex<double>`, and `complex<long double>` are literal types (??).

3 If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

4 If `z` is an lvalue of type *cv* `complex<T>` then:

(4.1) — the expression `reinterpret_cast<cv T(&)[2]>(z)` ~~shall be~~is well-formed,

(4.2) — `reinterpret_cast<cv T(&)[2]>(z)[0]` ~~shall designate~~designates the real part of `z`, and

(4.3) — `reinterpret_cast<cv T(&)[2]>(z)[1]` ~~shall designate~~designates the imaginary part of `z`.

Moreover, if `a` is an expression of type *cv* `complex<T>*` and the expression `a[i]` is well-defined for an integer expression `i`, then:

(4.4) — `reinterpret_cast<cv T*>(a)[2*i]` ~~shall designate~~designates the real part of `a[i]`, and

(4.5) — `reinterpret_cast<cv T*>(a)[2*i + 1]` ~~shall designate~~designates the imaginary part of `a[i]`.

26.4.1 Header <complex> synopsis [complex.syn]

```
namespace std {
  // 26.4.2, class template complex
  template<class T> class complex;
```

```
// 26.4.3, specializations
template<> class complex<float>;
template<> class complex<double>;
template<> class complex<long double>;

// 26.4.6, operators
template<class T> constexpr complex<T> operator+(const complex<T>&, const complex<T>&);
template<class T> constexpr complex<T> operator+(const complex<T>&, const T&);
template<class T> constexpr complex<T> operator+(const T&, const complex<T>&);

template<class T> constexpr complex<T> operator-(const complex<T>&, const complex<T>&);
template<class T> constexpr complex<T> operator-(const complex<T>&, const T&);
template<class T> constexpr complex<T> operator-(const T&, const complex<T>&);

template<class T> constexpr complex<T> operator*(const complex<T>&, const complex<T>&);
template<class T> constexpr complex<T> operator*(const complex<T>&, const T&);
template<class T> constexpr complex<T> operator*(const T&, const complex<T>&);

template<class T> constexpr complex<T> operator/(const complex<T>&, const complex<T>&);
template<class T> constexpr complex<T> operator/(const complex<T>&, const T&);
template<class T> constexpr complex<T> operator/(const T&, const complex<T>&);

template<class T> constexpr complex<T> operator+(const complex<T>&);
template<class T> constexpr complex<T> operator-(const complex<T>&);

template<class T> constexpr bool operator==(const complex<T>&, const complex<T>&);
template<class T> constexpr bool operator==(const complex<T>&, const T&);

template<class T, class charT, class traits>
  basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&, complex<T>&);

template<class T, class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, const complex<T>&);

// 26.4.7, values
template<class T> constexpr T real(const complex<T>&);
template<class T> constexpr T imag(const complex<T>&);

template<class T> T abs(const complex<T>&);
template<class T> T arg(const complex<T>&);
template<class T> constexpr T norm(const complex<T>&);

template<class T> constexpr complex<T> conj(const complex<T>&);
template<class T> complex<T> proj(const complex<T>&);
template<class T> complex<T> polar(const T&, const T& = T());

// 26.4.8, transcendentals
template<class T> complex<T> acos(const complex<T>&);
template<class T> complex<T> asin(const complex<T>&);
template<class T> complex<T> atan(const complex<T>&);

template<class T> complex<T> acosh(const complex<T>&);
template<class T> complex<T> asinh(const complex<T>&);
template<class T> complex<T> atanh(const complex<T>&);

template<class T> complex<T> cos  (const complex<T>&);
template<class T> complex<T> cosh (const complex<T>&);
template<class T> complex<T> exp  (const complex<T>&);
template<class T> complex<T> log  (const complex<T>&);
template<class T> complex<T> log10(const complex<T>&);

template<class T> complex<T> pow  (const complex<T>&, const T&);
template<class T> complex<T> pow  (const complex<T>&, const complex<T>&);
template<class T> complex<T> pow  (const T&, const complex<T>&);
```

```
  template<class T> complex<T> sin  (const complex<T>&);
  template<class T> complex<T> sinh (const complex<T>&);
  template<class T> complex<T> sqrt (const complex<T>&);
  template<class T> complex<T> tan  (const complex<T>&);
  template<class T> complex<T> tanh (const complex<T>&);

  // 26.4.10, complex literals
  inline namespace literals {
  inline namespace complex_literals {
    constexpr complex<long double> operator""il(long double);
    constexpr complex<long double> operator""il(unsigned long long);
    constexpr complex<double> operator""i(long double);
    constexpr complex<double> operator""i(unsigned long long);
    constexpr complex<float> operator""if(long double);
    constexpr complex<float> operator""if(unsigned long long);
  }
  }
}
```

26.4.2 Class template `complex` [complex]

```
namespace std {
  template<class T> class complex {
  public:
    using value_type = T;

    constexpr complex(const T& re = T(), const T& im = T());
    constexpr complex(const complex&);
    template<class X> constexpr complex(const complex<X>&);

    constexpr T real() const;
    constexpr void real(T);
    constexpr T imag() const;
    constexpr void imag(T);

    constexpr complex& operator= (const T&);
    constexpr complex& operator+=(const T&);
    constexpr complex& operator-=(const T&);
    constexpr complex& operator*=(const T&);
    constexpr complex& operator/=(const T&);

    constexpr complex& operator=(const complex&);
    template<class X> constexpr complex& operator= (const complex<X>&);
    template<class X> constexpr complex& operator+=(const complex<X>&);
    template<class X> constexpr complex& operator-=(const complex<X>&);
    template<class X> constexpr complex& operator*=(const complex<X>&);
    template<class X> constexpr complex& operator/=(const complex<X>&);
  };
}
```

1 The class `complex` describes an object that can store the Cartesian components, `real()` and `imag()`, of a complex number.

26.4.3 Specializations [complex.special]

```
namespace std {
  template<> class complex<float> {
  public:
    using value_type = float;

    constexpr complex(float re = 0.0f, float im = 0.0f);
    constexpr complex(const complex<float>&) = default;
    constexpr explicit complex(const complex<double>&);
    constexpr explicit complex(const complex<long double>&);
```

```
    constexpr float real() const;
    constexpr void real(float);
    constexpr float imag() const;
    constexpr void imag(float);

    constexpr complex& operator= (float);
    constexpr complex& operator+=(float);
    constexpr complex& operator-=(float);
    constexpr complex& operator*=(float);
    constexpr complex& operator/=(float);

    constexpr complex& operator=(const complex&);
    template<class X> constexpr complex& operator= (const complex<X>&);
    template<class X> constexpr complex& operator+=(const complex<X>&);
    template<class X> constexpr complex& operator-=(const complex<X>&);
    template<class X> constexpr complex& operator*=(const complex<X>&);
    template<class X> constexpr complex& operator/=(const complex<X>&);
  };

  template<> class complex<double> {
  public:
    using value_type = double;

    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
    constexpr complex(const complex<double>&) = default;
    constexpr explicit complex(const complex<long double>&);

    constexpr double real() const;
    constexpr void real(double);
    constexpr double imag() const;
    constexpr void imag(double);

    constexpr complex& operator= (double);
    constexpr complex& operator+=(double);
    constexpr complex& operator-=(double);
    constexpr complex& operator*=(double);
    constexpr complex& operator/=(double);

    constexpr complex& operator=(const complex&);
    template<class X> constexpr complex& operator= (const complex<X>&);
    template<class X> constexpr complex& operator+=(const complex<X>&);
    template<class X> constexpr complex& operator-=(const complex<X>&);
    template<class X> constexpr complex& operator*=(const complex<X>&);
    template<class X> constexpr complex& operator/=(const complex<X>&);
  };

  template<> class complex<long double> {
  public:
    using value_type = long double;

    constexpr complex(long double re = 0.0L, long double im = 0.0L);
    constexpr complex(const complex<float>&);
    constexpr complex(const complex<double>&);
    constexpr complex(const complex<long double>&) = default;

    constexpr long double real() const;
    constexpr void real(long double);
    constexpr long double imag() const;
    constexpr void imag(long double);

    constexpr complex& operator= (long double);
    constexpr complex& operator+=(long double);
    constexpr complex& operator-=(long double);
```

```
    constexpr complex& operator*=(long double);
    constexpr complex& operator/=(long double);

    constexpr complex& operator=(const complex&);
    template<class X> constexpr complex& operator= (const complex<X>&);
    template<class X> constexpr complex& operator+=(const complex<X>&);
    template<class X> constexpr complex& operator-=(const complex<X>&);
    template<class X> constexpr complex& operator*=(const complex<X>&);
    template<class X> constexpr complex& operator/=(const complex<X>&);
  };
}
```

26.4.4 Member functions [complex.members]

`template<class T> constexpr complex(const T& re = T(), const T& im = T());`

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

2 *Ensures:* `real() == re && imag() == im` is true.

`constexpr T real() const;`

3 *Returns:* The value of the real component.

`constexpr void real(T val);`

4 *Effects:* Assigns `val` to the real component.

`constexpr T imag() const;`

5 *Returns:* The value of the imaginary component.

`constexpr void imag(T val);`

6 *Effects:* Assigns `val` to the imaginary component.

26.4.5 Member operators [complex.member.ops]

`constexpr complex& operator+=(const T& rhs);`

1 *Effects:* Adds the scalar value `rhs` to the real part of the complex value `*this` and stores the result in the real part of `*this`, leaving the imaginary part unchanged.

2 *Returns:* `*this`.

`constexpr complex& operator-=(const T& rhs);`

3 *Effects:* Subtracts the scalar value `rhs` from the real part of the complex value `*this` and stores the result in the real part of `*this`, leaving the imaginary part unchanged.

4 *Returns:* `*this`.

`constexpr complex& operator*=(const T& rhs);`

5 *Effects:* Multiplies the scalar value `rhs` by the complex value `*this` and stores the result in `*this`.

6 *Returns:* `*this`.

`constexpr complex& operator/=(const T& rhs);`

7 *Effects:* Divides the scalar value `rhs` into the complex value `*this` and stores the result in `*this`.

8 *Returns:* `*this`.

`template<class X> constexpr complex& operator+=(const complex<X>& rhs);`

9 *Effects:* Adds the complex value `rhs` to the complex value `*this` and stores the sum in `*this`.

10 *Returns:* `*this`.

`template<class X> constexpr complex& operator-=(const complex<X>& rhs);`

11 *Effects:* Subtracts the complex value `rhs` from the complex value `*this` and stores the difference in `*this`.

12 *Returns:* `*this`.

```
template<class X> constexpr complex& operator*=(const complex<X>& rhs);
```

13 *Effects:* Multiplies the complex value `rhs` by the complex value `*this` and stores the product in `*this`.

14 *Returns:* `*this`.

```
template<class X> constexpr complex& operator/=(const complex<X>& rhs);
```

15 *Effects:* Divides the complex value `rhs` into the complex value `*this` and stores the quotient in `*this`.

16 *Returns:* `*this`.

26.4.6 Non-member operations [complex.ops]

```
template<class T> constexpr complex<T> operator+(const complex<T>& lhs);
```

1 *Returns:* `complex<T>(lhs)`.

2 *Remarks:* unary operator.

```
template<class T> constexpr complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs);
template<class T> constexpr complex<T> operator+(const complex<T>& lhs, const T& rhs);
template<class T> constexpr complex<T> operator+(const T& lhs, const complex<T>& rhs);
```

3 *Returns:* `complex<T>(lhs) += rhs`.

```
template<class T> constexpr complex<T> operator-(const complex<T>& lhs);
```

4 *Returns:* `complex<T>(-lhs.real(),-lhs.imag())`.

5 *Remarks:* unary operator.

```
template<class T> constexpr complex<T> operator-(const complex<T>& lhs, const complex<T>& rhs);
template<class T> constexpr complex<T> operator-(const complex<T>& lhs, const T& rhs);
template<class T> constexpr complex<T> operator-(const T& lhs, const complex<T>& rhs);
```

6 *Returns:* `complex<T>(lhs) -= rhs`.

```
template<class T> constexpr complex<T> operator*(const complex<T>& lhs, const complex<T>& rhs);
template<class T> constexpr complex<T> operator*(const complex<T>& lhs, const T& rhs);
template<class T> constexpr complex<T> operator*(const T& lhs, const complex<T>& rhs);
```

7 *Returns:* `complex<T>(lhs) *= rhs`.

```
template<class T> constexpr complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);
template<class T> constexpr complex<T> operator/(const complex<T>& lhs, const T& rhs);
template<class T> constexpr complex<T> operator/(const T& lhs, const complex<T>& rhs);
```

8 *Returns:* `complex<T>(lhs) /= rhs`.

```
template<class T> constexpr bool operator==(const complex<T>& lhs, const complex<T>& rhs);
template<class T> constexpr bool operator==(const complex<T>& lhs, const T& rhs);
```

9 *Returns:* `lhs.real() == rhs.real() && lhs.imag() == rhs.imag()`.

10 *Remarks:* The imaginary part is assumed to be `T()`, or 0.0, for the `T` arguments.

```
template<class T, class charT, class traits>
  basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& is, complex<T>& x);
```

[Editor's note: LWG 2714 proposes rewriting this entire description, so I've made a minimal change here.]

11 ~~*Requires:*~~ *Expects:* The input values ~~shall be~~are convertible to `T`.

12 *Effects:* Extracts a complex number `x` of the form: `u`, `(u)`, or `(u,v)`, where `u` is the real part and `v` is the imaginary part (**??**).

13 If bad input is encountered, calls `is.setstate(ios_base::failbit)` (which may throw `ios::failure` (**??**)).

14 *Returns:* `is`.

15 *Remarks:* This extraction is performed as a series of simpler extractions. Therefore, the skipping of whitespace is specified to be the same for each of the simpler extractions.

```
template<class T, class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& o, const complex<T>& x);
```

16 *Effects:* Inserts the complex number `x` onto the stream `o` as if it were implemented as follows:

```
basic_ostringstream<charT, traits> s;
s.flags(o.flags());
s.imbue(o.getloc());
s.precision(o.precision());
s << '(' << x.real() << "," << x.imag() << ')';
return o << s.str();
```

17 [*Note*: In a locale in which comma is used as a decimal point character, the use of comma as a field separator can be ambiguous. Inserting `showpoint` into the output stream forces all outputs to show an explicit decimal point character; as a result, all inserted sequences of complex numbers can be extracted unambiguously. — *end note*]

26.4.7 Value operations [complex.value.ops]

```
template<class T> constexpr T real(const complex<T>& x);
```

1 *Returns:* `x.real()`.

```
template<class T> constexpr T imag(const complex<T>& x);
```

2 *Returns:* `x.imag()`.

```
template<class T> T abs(const complex<T>& x);
```

3 *Returns:* The magnitude of `x`.

```
template<class T> T arg(const complex<T>& x);
```

4 *Returns:* The phase angle of `x`, or `atan2(imag(x), real(x))`.

```
template<class T> constexpr T norm(const complex<T>& x);
```

5 *Returns:* The squared magnitude of `x`.

```
template<class T> constexpr complex<T> conj(const complex<T>& x);
```

6 *Returns:* The complex conjugate of `x`.

```
template<class T> complex<T> proj(const complex<T>& x);
```

7 *Returns:* The projection of `x` onto the Riemann sphere.

8 *Remarks:* Behaves the same as the C function `cproj`. SEE ALSO: ISO C 7.3.9.5

```
template<class T> complex<T> polar(const T& rho, const T& theta = T());
```

9 ~~*Requires:*~~ *Expects:* `rho` ~~shall be~~is non-negative and non-NaN. `theta` ~~shall be~~is finite.

10 *Returns:* The `complex` value corresponding to a complex number whose magnitude is `rho` and whose phase angle is `theta`.

26.4.8 Transcendentals [complex.transcendentals]

```
template<class T> complex<T> acos(const complex<T>& x);
```

1 *Returns:* The complex arc cosine of `x`.

2 *Remarks:* Behaves the same as the C function `cacos`. SEE ALSO: ISO C 7.3.5.1

```
template<class T> complex<T> asin(const complex<T>& x);
```

3 *Returns:* The complex arc sine of `x`.

4 *Remarks:* Behaves the same as the C function `casin`. SEE ALSO: ISO C 7.3.5.2

```
template<class T> complex<T> atan(const complex<T>& x);
```

5 *Returns:* The complex arc tangent of `x`.

6 *Remarks:* Behaves the same as the C function `catan`. SEE ALSO: ISO C 7.3.5.3

```
template<class T> complex<T> acosh(const complex<T>& x);
```

7 *Returns:* The complex arc hyperbolic cosine of `x`.

8 *Remarks:* Behaves the same as the C function `cacosh`. SEE ALSO: ISO C 7.3.6.1

```
template<class T> complex<T> asinh(const complex<T>& x);
```

9 *Returns:* The complex arc hyperbolic sine of `x`.

10 *Remarks:* Behaves the same as the C function `casinh`. SEE ALSO: ISO C 7.3.6.2

```
template<class T> complex<T> atanh(const complex<T>& x);
```

11 *Returns:* The complex arc hyperbolic tangent of `x`.

12 *Remarks:* Behaves the same as the C function `catanh`. SEE ALSO: ISO C 7.3.6.3

```
template<class T> complex<T> cos(const complex<T>& x);
```

13 *Returns:* The complex cosine of `x`.

```
template<class T> complex<T> cosh(const complex<T>& x);
```

14 *Returns:* The complex hyperbolic cosine of `x`.

```
template<class T> complex<T> exp(const complex<T>& x);
```

15 *Returns:* The complex base-e exponential of `x`.

```
template<class T> complex<T> log(const complex<T>& x);
```

16 *Returns:* The complex natural (base-e) logarithm of `x`. For all `x`, `imag(log(x))` lies in the interval `[`$-\pi$`, `π`]`. [*Note*: The semantics of this function are intended to be the same in C++ as they are for `clog` in C. — *end note*]

17 *Remarks:* The branch cuts are along the negative real axis.

```
template<class T> complex<T> log10(const complex<T>& x);
```

18 *Returns:* The complex common (base-10) logarithm of `x`, defined as `log(x) / log(10)`.

19 *Remarks:* The branch cuts are along the negative real axis.

```
template<class T> complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> pow(const complex<T>& x, const T& y);
template<class T> complex<T> pow(const T& x, const complex<T>& y);
```

20 *Returns:* The complex power of base `x` raised to the $\mathtt{y}^{\text{th}}$ power, defined as `exp(y * log(x))`. The value returned for `pow(0, 0)` is implementation-defined.

21 *Remarks:* The branch cuts are along the negative real axis.

```
template<class T> complex<T> sin(const complex<T>& x);
```

22 *Returns:* The complex sine of `x`.

```
template<class T> complex<T> sinh(const complex<T>& x);
```

23 *Returns:* The complex hyperbolic sine of `x`.

```
template<class T> complex<T> sqrt(const complex<T>& x);
```

24 *Returns:* The complex square root of `x`, in the range of the right half-plane. [*Note*: The semantics of this function are intended to be the same in C++ as they are for `csqrt` in C. — *end note*]

25 *Remarks:* The branch cuts are along the negative real axis.

```
template<class T> complex<T> tan(const complex<T>& x);
```

26 *Returns:* The complex tangent of `x`.

```
template<class T> complex<T> tanh(const complex<T>& x);
```

27 *Returns:* The complex hyperbolic tangent of `x`.

26.4.9 Additional overloads [cmplx.over]

1 The following function templates shall have additional overloads:

```
arg                    norm
conj                   proj
imag                   real
```

where `norm`, `conj`, `imag`, and `real` are `constexpr` overloads.

2 The additional overloads shall be sufficient to ensure:

(2.1) — If the argument has type `long double`, then it is effectively cast to `complex<long double>`.

(2.2) — Otherwise, if the argument has type `double` or an integer type, then it is effectively cast to `complex<double>`.

(2.3) — Otherwise, if the argument has type `float`, then it is effectively cast to `complex<float>`.

3 Function template `pow` shall have additional overloads sufficient to ensure, for a call with at least one argument of type `complex<T>`:

(3.1) — If either argument has type `complex<long double>` or type `long double`, then both arguments are effectively cast to `complex<long double>`.

(3.2) — Otherwise, if either argument has type `complex<double>`, `double`, or an integer type, then both arguments are effectively cast to `complex<double>`.

(3.3) — Otherwise, if either argument has type `complex<float>` or `float`, then both arguments are effectively cast to `complex<float>`.

26.4.10 Suffixes for complex number literals [complex.literals]

1 This subclause describes literal suffixes for constructing complex number literals. The suffixes `i`, `il`, and `if` create complex numbers of the types `complex<double>`, `complex<long double>`, and `complex<float>` respectively, with their imaginary part denoted by the given literal number and the real part being zero.

```
constexpr complex<long double> operator""il(long double d);
constexpr complex<long double> operator""il(unsigned long long d);
```

2 *Returns:* `complex<long double>{0.0L, static_cast<long double>(d)}`.

```
constexpr complex<double> operator""i(long double d);
constexpr complex<double> operator""i(unsigned long long d);
```

3 *Returns:* `complex<double>{0.0, static_cast<double>(d)}`.

```
constexpr complex<float> operator""if(long double d);
constexpr complex<float> operator""if(unsigned long long d);
```

4 *Returns:* `complex<float>{0.0f, static_cast<float>(d)}`.

26.5 Bit manipulation [bit]

26.5.1 General [bit.general]

1 The header `<bit>` provides components to access, manipulate and process both individual bits and bit sequences.

26.5.2 Header `<bit>` synopsis [bit.syn]

```
namespace std {
  // 26.5.3, bit_cast
  template<class To, class From>
    constexpr To bit_cast(const From& from) noexcept;

  // 26.5.4, integral powers of 2
  template<class T>
    constexpr bool ispow2(T x) noexcept;
```

```
  template<class T>
    constexpr T ceil2(T x);
  template<class T>
    constexpr T floor2(T x) noexcept;
  template<class T>
    constexpr T log2p1(T x) noexcept;

  // 26.5.5, rotating
  template<class T>
    [[nodiscard]] constexpr T rotl(T x, int s) noexcept;
  template<class T>
    [[nodiscard]] constexpr T rotr(T x, int s) noexcept;

  // 26.5.6, counting
  template<class T>
    constexpr int countl_zero(T x) noexcept;
  template<class T>
    constexpr int countl_one(T x) noexcept;
  template<class T>
    constexpr int countr_zero(T x) noexcept;
  template<class T>
    constexpr int countr_one(T x) noexcept;
  template<class T>
    constexpr int popcount(T x) noexcept;

  // 26.5.7, endian
  enum class endian {
    little = see below,
    big    = see below,
    native = see below
  };
}
```

26.5.3 Function template `bit_cast` [bit.cast]

```
template<class To, class From>
  constexpr To bit_cast(const From& from) noexcept;
```

1 *Constraints:*

(1.1) — `sizeof(To) == sizeof(From)` is `true`;

(1.2) — `is_trivially_copyable_v<To>` is `true`; and

(1.3) — `is_trivially_copyable_v<From>` is `true`.

2 *Returns:* An object of type `To`. Each bit of the value representation of the result is equal to the corresponding bit in the object representation of `from`. Padding bits of the `To` object are unspecified. If there is no value of type `To` corresponding to the value representation produced, the behavior is undefined. If there are multiple such values, which value is produced is unspecified.

3 *Remarks:* This function shall not participate in overload resolution unless:

(3.1) — `sizeof(To) == sizeof(From)` is `true`;

(3.2) — `is_trivially_copyable_v<To>` is `true`; and

(3.3) — `is_trivially_copyable_v<From>` is `true`.

This function ~~shall be~~is `constexpr` if and only if `To`, `From`, and the types of all subobjects of `To` and `From` are types `T` such that:

(3.4) — `is_union_v<T>` is `false`;

(3.5) — `is_pointer_v<T>` is `false`;

(3.6) — `is_member_pointer_v<T>` is `false`;

(3.7) — `is_volatile_v<T>` is `false`; and

(3.8) — `T` has no non-static data members of reference type.

26.5.4 Integral powers of 2 [bit.pow.two]

```
template<class T>
  constexpr bool ispow2(T x) noexcept;
```

1 *Constraints:* `T` is an unsigned integer type (??).

2 *Returns:* `true` if `x` is an integral power of two; `false` otherwise.

3 *Remarks:* This function shall not participate in overload resolution unless `T` is an unsigned integer type (??).

```
template<class T>
  constexpr T ceil2(T x);
```

4 Let N be the smallest power of 2 greater than or equal to `x`.

5 *Constraints:* `T` is an unsigned integer type (??).

6 *Expects:* N is representable as a value of type `T`.

7 *Returns:* N.

8 *Throws:* Nothing.

9 *Remarks:* A function call expression that violates the precondition in the *Expects:* element is not a core constant expression (??).

```
template<class T>
  constexpr T floor2(T x) noexcept;
```

10 *Constraints:* `T` is an unsigned integer type (??).

11 *Returns:* If `x == 0`, `0`; otherwise the maximal value `y` such that `ispow2(y)` is `true` and `y <= x`.

12 *Remarks:* This function shall not participate in overload resolution unless `T` is an unsigned integer type (??).

```
template<class T>
  constexpr T log2p1(T x) noexcept;
```

13 *Constraints:* `T` is an unsigned integer type (??).

14 *Returns:* If `x == 0`, `0`; otherwise one plus the base-2 logarithm of `x`, with any fractional part discarded.

15 *Remarks:* This function shall not participate in overload resolution unless `T` is an unsigned integer type (??).

26.5.5 Rotating [bit.rotate]

In the following descriptions, let `N` denote `numeric_limits<T>::digits`.

```
template<class T>
  [[nodiscard]] constexpr T rotl(T x, int s) noexcept;
```

1 *Constraints:* `T` is an unsigned integer type (??).

2 Let `r` be `s % N`.

3 *Returns:* If `r` is `0`, `x`; if `r` is positive, `(x << r) | (x >> (N - r))`; if `r` is negative, `rotr(x, -r)`.

```
template<class T>
  [[nodiscard]] constexpr T rotr(T x, int s) noexcept;
```

4 *Constraints:* `T` is an unsigned integer type (??).

5 Let `r` be `s % N`.

6 *Returns:* If `r` is `0`, `x`; if `r` is positive, `(x >> r) | (x << (N - r))`; if `r` is negative, `rotl(x, -r)`.

26.5.6 Counting [bit.count]

In the following descriptions, let `N` denote `numeric_limits<T>::digits`.

```
template<class T>
  constexpr int countl_zero(T x) noexcept;
```

1 *Constraints:* `T` is an unsigned integer type (??).

2 *Returns:* The number of consecutive `0` bits in the value of `x`, starting from the most significant bit. [*Note*: Returns `N` if `x == 0`. — *end note*]

```
template<class T>
  constexpr int countl_one(T x) noexcept;
```

3 *Constraints:* `T` is an unsigned integer type (??).

4 *Returns:* The number of consecutive `1` bits in the value of `x`, starting from the most significant bit. [*Note*: Returns `N` if `x == numeric_limits<T>::max()`. — *end note*]

```
template<class T>
  constexpr int countr_zero(T x) noexcept;
```

5 *Constraints:* `T` is an unsigned integer type (??).

6 *Returns:* The number of consecutive `0` bits in the value of `x`, starting from the least significant bit. [*Note*: Returns `N` if `x == 0`. — *end note*]

```
template<class T>
  constexpr int countr_one(T x) noexcept;
```

7 *Constraints:* `T` is an unsigned integer type (??).

8 *Returns:* The number of consecutive `1` bits in the value of `x`, starting from the least significant bit. [*Note*: Returns `N` if `x == numeric_limits<T>::max()`. — *end note*]

```
template<class T>
  constexpr int popcount(T x) noexcept;
```

9 *Constraints:* `T` is an unsigned integer type (??).

10 *Returns:* The number of `1` bits in the value of `x`.

26.5.7 Endian [bit.endian]

1 Two common methods of byte ordering in multibyte scalar types are big-endian and little-endian in the execution environment. Big-endian is a format for storage of binary data in which the most significant byte is placed first, with the rest in descending order. Little-endian is a format for storage of binary data in which the least significant byte is placed first, with the rest in ascending order. This subclause describes the endianness of the scalar types of the execution environment.

```
enum class endian {
  little = see below,
  big    = see below,
  native = see below
};
```

2 If all scalar types have size 1 byte, then all of `endian::little`, `endian::big`, and `endian::native` have the same value. Otherwise, `endian::little` is not equal to `endian::big`. If all scalar types are big-endian, `endian::native` is equal to `endian::big`. If all scalar types are little-endian, `endian::native` is equal to `endian::little`. Otherwise, `endian::native` is not equal to either `endian::big` or `endian::little`.

26.6 Random number generation [rand]

1 This subclause defines a facility for generating (pseudo-)random numbers.

2 In addition to a few utilities, four categories of entities are described: *uniform random bit generators*, *random number engines*, *random number engine adaptors*, and *random number distributions*. These categorizations are applicable to types that meet the corresponding requirements, to objects instantiated from such types, and to templates producing such types when instantiated. [*Note*: These entities are specified in such a way as to permit the binding of any uniform random bit generator object `e` as the argument to any random number distribution object `d`, thus producing a zero-argument function object such as given by `bind(d,e)`. — *end note*]

3 Each of the entities specified via this subclause has an associated arithmetic type (??) identified as `result_type`. With `T` as the `result_type` thus associated with such an entity, that entity is characterized:

(3.1) — as *boolean* or equivalently as *boolean-valued*, if `T` is `bool`;

(3.2) — otherwise as *integral* or equivalently as *integer-valued*, if `numeric_limits<T>::is_integer` is `true`;

(3.3) — otherwise as *floating* or equivalently as *real-valued*.

If integer-valued, an entity may optionally be further characterized as *signed* or *unsigned*, according to `numeric_limits<T>::is_signed`.

4 Unless otherwise specified, all descriptions of calculations in this subclause use mathematical real numbers.

5 Throughout this subclause, the operators bitand, bitor, and xor denote the respective conventional bitwise operations. Further:

(5.1) — the operator rshift denotes a bitwise right shift with zero-valued bits appearing in the high bits of the result, and

(5.2) — the operator lshift_w denotes a bitwise left shift with zero-valued bits appearing in the low bits of the result, and whose result is always taken modulo 2^w.

26.6.1 Header `<random>` synopsis [rand.synopsis]

```
#include <initializer_list>

namespace std {
  // 26.6.2.3, uniform random bit generator requirements
  template<class G>
    concept uniform_random_bit_generator = see below;

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

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

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

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

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

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

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

  using default_random_engine = see below;
```

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

// 26.6.7.1, class seed_seq
class seed_seq;

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

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

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

// 26.6.8.3.1, class bernoulli_distribution
class bernoulli_distribution;

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

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

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

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

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

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

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

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

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

// 26.6.8.5.2, class template lognormal_distribution
template<class RealType = double>
  class lognormal_distribution;

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

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

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

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

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

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

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

26.6.2 Requirements [rand.req]

26.6.2.1 General requirements [rand.req.genl]

1 Throughout this subclause 26.6, the effect of instantiating a template:

(1.1) — that has a template type parameter named `Sseq` is undefined unless the corresponding template argument is cv-unqualified and meets the requirements of seed sequence (26.6.2.2).

(1.2) — that has a template type parameter named `URBG` is undefined unless the corresponding template argument is cv-unqualified and meets the requirements of uniform random bit generator (26.6.2.3).

(1.3) — that has a template type parameter named `Engine` is undefined unless the corresponding template argument is cv-unqualified and meets the requirements of random number engine (26.6.2.4).

(1.4) — that has a template type parameter named `RealType` is undefined unless the corresponding template argument is cv-unqualified and is one of `float`, `double`, or `long double`.

(1.5) — that has a template type parameter named `IntType` is undefined unless the corresponding template argument is cv-unqualified and is one of `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

(1.6) — that has a template type parameter named `UIntType` is undefined unless the corresponding template argument is cv-unqualified and is one of `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

2 Throughout this subclause 26.6, phrases of the form "`x` is an iterator of a specific kind" shall be interpreted as equivalent to the more formal requirement that "`x` is a value of a type meeting the requirements of the specified iterator type".

3 Throughout this subclause 26.6, any constructor that can be called with a single argument and that meets a requirement specified in this subclause shall be declared `explicit`.

26.6.2.2 Seed sequence requirements [rand.req.seedseq]

1 A *seed sequence* is an object that consumes a sequence of integer-valued data and produces a requested number of unsigned integer values i, $0 \leq i < 2^{32}$, based on the consumed data. [*Note*: Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful, for example, in applications requiring large numbers of random number engines. — *end note*]

2 A class `S` meets the requirements of a seed sequence if the expressions shown in Table 92 are valid and have the indicated semantics, and if `S` also meets all other requirements of this subclause 26.6.2.2. In that Table and throughout this subclause:

(2.1) — `T` is the type named by `S`'s associated `result_type`;

(2.2) — q is a value of S and r is a possibly const value of S;

(2.3) — ib and ie are input iterators with an unsigned integer value_type of at least 32 bits;

(2.4) — rb and re are mutable random access iterators with an unsigned integer value_type of at least 32 bits;

(2.5) — ob is an output iterator; and

(2.6) — il is a value of initializer_list<T>.

Table 92: Seed sequence requirements [tab:rand.req.seedseq]

Expression	Return type	Pre/post-condition	Complexity
S::result_type	T	T is an unsigned integer type (??) of at least 32 bits.	compile-time
S()		Creates a seed sequence with the same initial state as all other default-constructed seed sequences of type S.	constant
S(ib,ie)		Creates a seed sequence having internal state that depends on some or all of the bits of the supplied sequence [ib, ie).	$\mathcal{O}$(ie − ib)
S(il)		Same as S(il.begin(), il.end()).	same as S(il.begin(), il.end())
q.generate(rb,re)	void	Does nothing if rb == re. Otherwise, fills the supplied sequence [rb, re) with 32-bit quantities that depend on the sequence supplied to the constructor and possibly also depend on the history of generate's previous invocations.	$\mathcal{O}$(re − rb)
r.size()	size_t	The number of 32-bit units that would be copied by a call to r.param.	constant
r.param(ob)	void	Copies to the given destination a sequence of 32-bit units that can be provided to the constructor of a second object of type S, and that would reproduce in that second object a state indistinguishable from the state of the first object.	$\mathcal{O}$(r.size())

26.6.2.3 Uniform random bit generator requirements [rand.req.urng]

1 A *uniform random bit generator* g of type G is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. [*Note*: The degree to which g's results approximate the ideal is often determined statistically. — *end note*]

```
template<class G>
  concept uniform_random_bit_generator =
    invocable<G&> && unsigned_integral<invoke_result_t<G&>> &&
    requires {
      { G::min() } -> same_as<invoke_result_t<G&>>;
      { G::max() } -> same_as<invoke_result_t<G&>>;
    };
```

2 Let g be an object of type G. G models uniform_random_bit_generator only if

(2.1) — both G::min() and G::max() are constant expressions (??), (2.2) — `G::min() < G::max()`,

(2.3) — `G::min() <= g()`,

(2.4) — `g() <= G::max()`, and

(2.5) — `g()` has amortized constant complexity.

3 A class `G` meets the *uniform random bit generator* requirements if `G` models `uniform_random_bit_generator`, `invoke_result_t<G&>` is an unsigned integer type (**??**), and `G` provides a nested *typedef-name* `result_type` that denotes the same type as `invoke_result_t<G&>`.

26.6.2.4 Random number engine requirements [rand.req.eng]

1 A *random number engine* (commonly shortened to *engine*) `e` of type `E` is a uniform random bit generator that additionally meets the requirements (e.g., for seeding and for input/output) specified in this subclause.

2 At any given time, `e` has a state $\mathtt{e}_i$ for some integer $i \geq 0$. Upon construction, `e` has an initial state $\mathtt{e}_0$. An engine's state may be established via a constructor, a `seed` function, assignment, or a suitable `operator>>`.

3 `E`'s specification shall define:

(3.1) — the size of `E`'s state in multiples of the size of `result_type`, given as an integral constant expression;

(3.2) — the *transition algorithm* TA by which `e`'s state $\mathtt{e}_i$ is advanced to its *successor state* $\mathtt{e}_{i+1}$; and

(3.3) — the *generation algorithm* GA by which an engine's state is mapped to a value of type `result_type`.

4 A class `E` that meets the requirements of a uniform random bit generator (26.6.2.3) also meets the requirements of a *random number engine* if the expressions shown in Table 93 are valid and have the indicated semantics, and if `E` also meets all other requirements of this subclause 26.6.2.4. In that Table and throughout this subclause:

(4.1) — `T` is the type named by `E`'s associated `result_type`;

(4.2) — `e` is a value of `E`, `v` is an lvalue of `E`, `x` and `y` are (possibly `const`) values of `E`;

(4.3) — `s` is a value of `T`;

(4.4) — `q` is an lvalue meeting the requirements of a seed sequence (26.6.2.2);

(4.5) — `z` is a value of type `unsigned long long`;

(4.6) — `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`; and

(4.7) — `is` is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`;

where `charT` and `traits` are constrained according to **??** and **??**.

Table 93: Random number engine requirements [tab:rand.req.eng]

Expression	Return type	Pre/post-condition	Complexity
`E()`		Creates an engine with the same initial state as all other default-constructed engines of type `E`.	$\mathcal{O}$(size of state)
`E(x)`		Creates an engine that compares equal to `x`.	$\mathcal{O}$(size of state)
`E(s)`		Creates an engine with initial state determined by `s`.	$\mathcal{O}$(size of state)
`E(q)`[242]		Creates an engine with an initial state that depends on a sequence produced by one call to `q.generate`.	same as complexity of `q.generate` called on a sequence whose length is size of state
`e.seed()`	`void`	*Ensures:* `e == E()`.	same as `E()`
`e.seed(s)`	`void`	*Ensures:* `e == E(s)`.	same as `E(s)`

242) This constructor (as well as the subsequent corresponding `seed()` function) may be particularly useful to applications requiring a large number of independent random sequences.

Expression	Return type	Pre/post-condition	Complexity
`e.seed(q)`	`void`	*Ensures:* `e == E(q)`.	same as `E(q)`
`e()`	`T`	Advances `e`'s state e_i to e_{i+1} $= \mathsf{TA}(\mathsf{e}_i)$ and returns $\mathsf{GA}(\mathsf{e}_i)$.	per 26.6.2.3
`e.discard(z)` [243]	`void`	Advances `e`'s state e_i to $\mathsf{e}_{i+\mathtt{z}}$ by any means equivalent to `z` consecutive calls `e()`.	no worse than the complexity of `z` consecutive calls `e()`
`x == y`	`bool`	This operator is an equivalence relation. With S_x and S_y as the infinite sequences of values that would be generated by repeated future calls to `x()` and `y()`, respectively, returns `true` if $S_x = S_y$; else returns `false`.	$\mathscr{O}$(size of state)
`x != y`	`bool`	`!(x == y)`.	$\mathscr{O}$(size of state)
`os << x`	reference to the type of `os`	With `os.`*fmtflags* set to `ios_base::dec|ios_base::left` and the fill character set to the space character, writes to `os` the textual representation of `x`'s current state. In the output, adjacent numbers are separated by one or more space characters. *Ensures:* The `os.`*fmtflags* and fill character are unchanged.	$\mathscr{O}$(size of state)
`is >> v`	reference to the type of `is`	With `is.fmtflags` set to `ios_base::dec`, sets `v`'s state as determined by reading its textual representation from `is`. If bad input is encountered, ensures that `v`'s state is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `ios::failure` (??)). If a textual representation written via `os << x` was subsequently read via `is >> v`, then `x == v` provided that there have been no intervening invocations of `x` or of `v`. ~~*Requires:*~~ *Expects:* `is` provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of `is`, and whose type's template specialization arguments `charT` and `traits` were respectively the same as those of `is`. *Ensures:* The `is.`*fmtflags* are unchanged.	$\mathscr{O}$(size of state)

243) This operation is common in user code, and can often be implemented in an engine-specific manner so as to provide significant performance improvements over an equivalent naive loop that makes `z` consecutive calls `e()`.

5 `E` shall meet the *Cpp17CopyConstructible* (Table **??**) and *Cpp17CopyAssignable* (Table **??**) requirements. These operations shall each be of complexity no worse than $\mathcal{O}$(size of state).

26.6.2.5 Random number engine adaptor requirements [rand.req.adapt]

1 A *random number engine adaptor* (commonly shortened to *adaptor*) `a` of type `A` is a random number engine that takes values produced by some other random number engine, and applies an algorithm to those values in order to deliver a sequence of values with different randomness properties. An engine `b` of type `B` adapted in this way is termed a *base engine* in this context. The expression `a.base()` shall be valid and shall return a const reference to `a`'s base engine.

2 The requirements of a random number engine type shall be interpreted as follows with respect to a random number engine adaptor type.

```
A::A();
```

3 *Effects:* The base engine is initialized as if by its default constructor.

```
bool operator==(const A& a1, const A& a2);
```

4 *Returns:* `true` if `a1`'s base engine is equal to `a2`'s base engine. Otherwise returns `false`.

```
A::A(result_type s);
```

5 *Effects:* The base engine is initialized with `s`.

```
template<class Sseq> A::A(Sseq& q);
```

6 *Effects:* The base engine is initialized with `q`.

```
void seed();
```

7 *Effects:* With `b` as the base engine, invokes `b.seed()`.

```
void seed(result_type s);
```

8 *Effects:* With `b` as the base engine, invokes `b.seed(s)`.

```
template<class Sseq> void seed(Sseq& q);
```

9 *Effects:* With `b` as the base engine, invokes `b.seed(q)`.

10 `A` shall also meet the following additional requirements:

(10.1) — The complexity of each function shall not exceed the complexity of the corresponding function applied to the base engine.

(10.2) — The state of `A` shall include the state of its base engine. The size of `A`'s state shall be no less than the size of the base engine.

(10.3) — Copying `A`'s state (e.g., during copy construction or copy assignment) shall include copying the state of the base engine of `A`.

(10.4) — The textual representation of `A` shall include the textual representation of its base engine.

26.6.2.6 Random number distribution requirements [rand.req.dist]

1 A *random number distribution* (commonly shortened to *distribution*) `d` of type `D` is a function object returning values that are distributed according to an associated mathematical *probability density function* $p(z)$ or according to an associated *discrete probability function* $P(z_i)$. A distribution's specification identifies its associated probability function $p(z)$ or $P(z_i)$.

2 An associated probability function is typically expressed using certain externally-supplied quantities known as the *parameters of the distribution*. Such distribution parameters are identified in this context by writing, for example, $p(z \,|\, a, b)$ or $P(z_i \,|\, a, b)$, to name specific parameters, or by writing, for example, $p(z \,|\, \{\texttt{p}\})$ or $P(z_i \,|\, \{\texttt{p}\})$, to denote a distribution's parameters `p` taken as a whole.

3 A class `D` meets the requirements of a *random number distribution* if the expressions shown in Table 94 are valid and have the indicated semantics, and if `D` and its associated types also meet all other requirements of this subclause 26.6.2.6. In that Table and throughout this subclause,

(3.1) — `T` is the type named by `D`'s associated `result_type`;

(3.2) — `P` is the type named by `D`'s associated `param_type`;

(3.3) — `d` is a value of `D`, and `x` and `y` are (possibly `const`) values of `D`;

(3.4) — `glb` and `lub` are values of `T` respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by `d`'s `operator()`, as determined by the current values of `d`'s parameters;

(3.5) — `p` is a (possibly `const`) value of `P`;

(3.6) — `g`, `g1`, and `g2` are lvalues of a type meeting the requirements of a uniform random bit generator (26.6.2.3);

(3.7) — `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`; and

(3.8) — `is` is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`;

where `charT` and `traits` are constrained according to ?? and ??.

Table 94: Random number distribution requirements [tab:rand.req.dist]

Expression	Return type	Pre/post-condition	Complexity
`D::result_type`	`T`	`T` is an arithmetic type (??).	compile-time
`D::param_type`	`P`		compile-time
`D()`		Creates a distribution whose behavior is indistinguishable from that of any other newly default-constructed distribution of type `D`.	constant
`D(p)`		Creates a distribution whose behavior is indistinguishable from that of a distribution newly constructed directly from the values used to construct `p`.	same as `p`'s construction
`d.reset()`	`void`	Subsequent uses of `d` do not depend on values produced by any engine prior to invoking `reset`.	constant
`x.param()`	`P`	Returns a value `p` such that `D(p).param() == p`.	no worse than the complexity of `D(p)`
`d.param(p)`	`void`	*Ensures:* `d.param() == p`.	no worse than the complexity of `D(p)`
`d(g)`	`T`	With `p` = `d.param()`, the sequence of numbers returned by successive invocations with the same object `g` is randomly distributed according to the associated $p(z \mid \{\mathtt{p}\})$ or $P(z_i \mid \{\mathtt{p}\})$ function.	amortized constant number of invocations of `g`
`d(g,p)`	`T`	The sequence of numbers returned by successive invocations with the same objects `g` and `p` is randomly distributed according to the associated $p(z \mid \{\mathtt{p}\})$ or $P(z_i \mid \{\mathtt{p}\})$ function.	amortized constant number of invocations of `g`
`x.min()`	`T`	Returns `glb`.	constant
`x.max()`	`T`	Returns `lub`.	constant

Expression	Return type	Pre/post-condition	Complexity
`x == y`	`bool`	This operator is an equivalence relation. Returns `true` if `x.param() == y.param()` and $S_1 = S_2$, where S_1 and S_2 are the infinite sequences of values that would be generated, respectively, by repeated future calls to `x(g1)` and `y(g2)` whenever `g1 == g2`. Otherwise returns `false`.	constant
`x != y`	`bool`	`!(x == y)`.	same as `x == y`.
`os << x`	reference to the type of `os`	Writes to `os` a textual representation for the parameters and the additional internal data of `x`. *Ensures:* The `os.`*fmtflags* and fill character are unchanged.	
`is >> d`	reference to the type of `is`	Restores from `is` the parameters and additional internal data of the lvalue `d`. If bad input is encountered, ensures that `d` is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `ios::failure` (??)). ~~*Requires:*~~ *Expects:* `is` provides a textual representation that was previously written using an `os` whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same as those of `is`. *Ensures:* The `is.`*fmtflags* are unchanged.	

4 `D` shall meet the *Cpp17CopyConstructible* (Table **??**) and *Cpp17CopyAssignable* (Table **??**) requirements.

5 The sequence of numbers produced by repeated invocations of `d(g)` shall be independent of any invocation of `os << d` or of any `const` member function of `D` between any of the invocations `d(g)`.

6 If a textual representation is written using `os << x` and that representation is restored into the same or a different object `y` of the same type using `is >> y`, repeated invocations of `y(g)` shall produce the same sequence of numbers as would repeated invocations of `x(g)`.

7 It is unspecified whether `D::param_type` is declared as a (nested) `class` or via a `typedef`. In this subclause 26.6, declarations of `D::param_type` are in the form of `typedef`s for convenience of exposition only.

8 `P` shall meet the *Cpp17CopyConstructible* (Table **??**), *Cpp17CopyAssignable* (Table **??**), and *Cpp17Equality-Comparable* (Table **??**) requirements.

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

10 `P` shall have a declaration of the form

```
using distribution_type =  D;
```

26.6.3 Random number engine class templates [rand.eng]

1 Each type instantiated from a class template specified in this subclause 26.6.3 meets the requirements of a random number engine (26.6.2.4) type.

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

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

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

5 Descriptions are provided in this subclause 26.6.3 only for engine operations that are not described in 26.6.2.4 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

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

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

(7.1) — if the constructor

```
template<class Sseq> explicit X(Sseq& q);
```

is called with a type `Sseq` that does not qualify as a seed sequence, then this constructor shall not participate in overload resolution;

(7.2) — if the member function

```
template<class Sseq> void seed(Sseq& q);
```

is called with a type `Sseq` that does not qualify as a seed sequence, then this function shall not participate in overload resolution.

The extent to which an implementation determines that a type cannot be a seed sequence is unspecified, except that as a minimum a type shall not qualify as a seed sequence if it is implicitly convertible to `X::result_type`.

26.6.3.1 Class template `linear_congruential_engine` [rand.eng.lcong]

1 A `linear_congruential_engine` random number engine produces unsigned integer random numbers. The state $\mathtt{x}_i$ of a `linear_congruential_engine` object `x` is of size 1 and consists of a single integer. The transition algorithm is a modular linear function of the form $\mathsf{TA}(\mathtt{x}_i) = (a \cdot \mathtt{x}_i + c) \bmod m$; the generation algorithm is $\mathsf{GA}(\mathtt{x}_i) = \mathtt{x}_{i+1}$.

```
template<class UIntType, UIntType a, UIntType c, UIntType m>
  class linear_congruential_engine {
  public:
    // types
    using result_type = UIntType;

    // engine characteristics
    static constexpr result_type multiplier = a;
    static constexpr result_type increment = c;
    static constexpr result_type modulus = m;
    static constexpr result_type min() { return c == 0u ? 1u: 0u; }
    static constexpr result_type max() { return m - 1u; }
    static constexpr result_type default_seed = 1u;

    // constructors and seeding functions
    linear_congruential_engine() : linear_congruential_engine(default_seed) {}
    explicit linear_congruential_engine(result_type s);
    template<class Sseq> explicit linear_congruential_engine(Sseq& q);
    void seed(result_type s = default_seed);
    template<class Sseq> void seed(Sseq& q);
```

```
    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
  };
```

2 If the template parameter `m` is 0, the modulus m used throughout this subclause 26.6.3.1 is `numeric_limits<result_type>::max()` plus 1. [*Note*: m need not be representable as a value of type `result_type`. — *end note*]

3 If the template parameter `m` is not 0, the following relations shall hold: `a < m` and `c < m`.

4 The textual representation consists of the value of $\mathtt{x}_i$.

```
explicit linear_congruential_engine(result_type s);
```

5 *Effects:* ~~Constructs a `linear_congruential_engine` object.~~ If $c \bmod m$ is 0 and `s` $\bmod m$ is 0, sets the engine's state to 1, otherwise sets the engine's state to `s` $\bmod m$.

```
template<class Sseq> explicit linear_congruential_engine(Sseq& q);
```

6 *Effects:* ~~Constructs a `linear_congruential_engine` object.~~ With $k = \left\lceil \frac{\log_2 m}{32} \right\rceil$ and a an array (or equivalent) of length $k+3$, invokes `q.generate(`$a+0$`, `$a+k+3$`)` and then computes $S = \left(\sum_{j=0}^{k-1} a_{j+3} \cdot 2^{32j}\right) \bmod m$. If $c \bmod m$ is 0 and S is 0, sets the engine's state to 1, else sets the engine's state to S.

26.6.3.2 Class template `mersenne_twister_engine` [rand.eng.mers]

1 A `mersenne_twister_engine` random number engine[244] produces unsigned integer random numbers in the closed interval $[0, 2^w - 1]$. The state $\mathtt{x}_i$ of a `mersenne_twister_engine` object `x` is of size n and consists of a sequence X of n values of the type delivered by `x`; all subscripts applied to X are to be taken modulo n.

2 The transition algorithm employs a twisted generalized feedback shift register defined by shift values n and m, a twist value r, and a conditional xor-mask a. To improve the uniformity of the result, the bits of the raw shift register are additionally *tempered* (i.e., scrambled) according to a bit-scrambling matrix defined by values u, d, s, b, t, c, and ℓ.

The state transition is performed as follows:

(2.1) — Concatenate the upper $w - r$ bits of X_{i-n} with the lower r bits of X_{i+1-n} to obtain an unsigned integer value Y.

(2.2) — With $\alpha = a \cdot (Y \mathsf{~bitand~} 1)$, set X_i to $X_{i+m-n} \mathsf{~xor~} (Y \mathsf{~rshift~} 1) \mathsf{~xor~} \alpha$.

The sequence X is initialized with the help of an initialization multiplier f.

3 The generation algorithm determines the unsigned integer values z_1, z_2, z_3, z_4 as follows, then delivers z_4 as its result:

(3.1) — Let $z_1 = X_i \mathsf{~xor~} \big((X_i \mathsf{~rshift~} u) \mathsf{~bitand~} d\big)$.

(3.2) — Let $z_2 = z_1 \mathsf{~xor~} \big((z_1 \mathsf{~lshift}_w~ s) \mathsf{~bitand~} b\big)$.

(3.3) — Let $z_3 = z_2 \mathsf{~xor~} \big((z_2 \mathsf{~lshift}_w~ t) \mathsf{~bitand~} c\big)$.

(3.4) — Let $z_4 = z_3 \mathsf{~xor~} (z_3 \mathsf{~rshift~} \ell)$.

```
template<class UIntType, size_t w, size_t n, size_t m, size_t r,
         UIntType a, size_t u, UIntType d, size_t s,
         UIntType b, size_t t,
         UIntType c, size_t l, UIntType f>
  class mersenne_twister_engine {
  public:
    // types
    using result_type = UIntType;

    // engine characteristics
    static constexpr size_t word_size = w;
    static constexpr size_t state_size = n;
```

244) The name of this engine refers, in part, to a property of its period: For properly-selected values of the parameters, the period is closely related to a large Mersenne prime number.

```
    static constexpr size_t shift_size = m;
    static constexpr size_t mask_bits = r;
    static constexpr UIntType xor_mask = a;
    static constexpr size_t tempering_u = u;
    static constexpr UIntType tempering_d = d;
    static constexpr size_t tempering_s = s;
    static constexpr UIntType tempering_b = b;
    static constexpr size_t tempering_t = t;
    static constexpr UIntType tempering_c = c;
    static constexpr size_t tempering_l = l;
    static constexpr UIntType initialization_multiplier = f;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return  2^w − 1; }
    static constexpr result_type default_seed = 5489u;

    // constructors and seeding functions
    mersenne_twister_engine() : mersenne_twister_engine(default_seed) {}
    explicit mersenne_twister_engine(result_type value);
    template<class Sseq> explicit mersenne_twister_engine(Sseq& q);
    void seed(result_type value = default_seed);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
  };
```

4 The following relations shall hold: `0 < m`, `m <= n`, `2u < w`, `r <= w`, `u <= w`, `s <= w`, `t <= w`, `l <= w`, `w <= numeric_limits<UIntType>::digits`, `a <= (1u<<w) - 1u`, `b <= (1u<<w) - 1u`, `c <= (1u<<w) - 1u`, `d <= (1u<<w) - 1u`, and `f <= (1u<<w) - 1u`.

5 The textual representation of $\mathtt{x}_i$ consists of the values of $X_{i-n}, \ldots, X_{i-1}$, in that order.

`explicit mersenne_twister_engine(result_type value);`

6 *Effects:* ~~Constructs a `mersenne_twister_engine` object.~~ Sets X_{-n} to `value` mod 2^w. Then, iteratively for $i = 1 - n, \ldots, -1$, sets X_i to

$$\left[f \cdot \left(X_{i-1} \ \mathsf{xor} \ \left(X_{i-1} \ \mathsf{rshift} \ (w-2)\right)\right) + i \bmod n\right] \bmod 2^w .$$

7 *Complexity:* $\mathscr{O}(n)$.

`template<class Sseq> explicit mersenne_twister_engine(Sseq& q);`

8 *Effects:* ~~Constructs a `mersenne_twister_engine` object.~~ With $k = \lceil w/32 \rceil$ and a an array (or equivalent) of length $n \cdot k$, invokes `q.generate(`$a+0$`, `$a+n\cdot k$`)` and then, iteratively for $i = -n, \ldots, -1$, sets X_i to $\left(\sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32j}\right) \bmod 2^w$. Finally, if the most significant $w - r$ bits of X_{-n} are zero, and if each of the other resulting X_i is 0, changes X_{-n} to 2^{w-1}.

26.6.3.3 Class template `subtract_with_carry_engine` [rand.eng.sub]

1 A `subtract_with_carry_engine` random number engine produces unsigned integer random numbers.

2 The state $\mathtt{x}_i$ of a `subtract_with_carry_engine` object `x` is of size $\mathscr{O}(r)$, and consists of a sequence X of r integer values $0 \le X_i < m = 2^w$; all subscripts applied to X are to be taken modulo r. The state $\mathtt{x}_i$ additionally consists of an integer c (known as the *carry*) whose value is either 0 or 1.

3 The state transition is performed as follows:

(3.1) — Let $Y = X_{i-s} - X_{i-r} - c$.

(3.2) — Set X_i to $y = Y \bmod m$. Set c to 1 if $Y < 0$, otherwise set c to 0.

[*Note*: This algorithm corresponds to a modular linear function of the form $\mathsf{TA}(\mathtt{x}_i) = (a \cdot \mathtt{x}_i) \bmod b$, where b is of the form $m^r - m^s + 1$ and $a = b - (b-1)/m$. — *end note*]

4 The generation algorithm is given by $\mathsf{GA}(\mathtt{x}_i) = y$, where y is the value produced as a result of advancing the engine's state as described above.

```
template<class UIntType, size_t w, size_t s, size_t r>
  class subtract_with_carry_engine {
  public:
    // types
    using result_type = UIntType;

    // engine characteristics
    static constexpr size_t word_size = w;
    static constexpr size_t short_lag = s;
    static constexpr size_t long_lag = r;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return m − 1; }
    static constexpr result_type default_seed = 19780503u;

    // constructors and seeding functions
    subtract_with_carry_engine() : subtract_with_carry_engine(default_seed) {}
    explicit subtract_with_carry_engine(result_type value);
    template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);
    void seed(result_type value = default_seed);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
  };
```

5 The following relations shall hold: `0u < s`, `s < r`, `0 < w`, and `w <= numeric_limits<UIntType>::digits`.

6 The textual representation consists of the values of $X_{i-r}, \dots, X_{i-1}$, in that order, followed by c.

`explicit subtract_with_carry_engine(result_type value);`

7 *Effects:* ~~Constructs a `subtract_with_carry_engine` object.~~ Sets the values of $X_{-r}, \dots, X_{-1}$, in that order, as specified below. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

To set the values X_k, first construct `e`, a `linear_congruential_engine` object, as if by the following definition:

```
linear_congruential_engine<result_type,
                           40014u,0u,2147483563u> e(value == 0u ? default_seed : value);
```

Then, to set each X_k, obtain new values $z_0, \dots, z_{n-1}$ from $n = \lceil w/32 \rceil$ successive invocations of `e` taken modulo 2^{32}. Set X_k to $\left(\sum_{j=0}^{n-1} z_j \cdot 2^{32j}\right) \bmod m$.

8 *Complexity:* Exactly $n \cdot$ `r` invocations of `e`.

`template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);`

9 *Effects:* ~~Constructs a `subtract_with_carry_engine` object.~~ With $k = \lceil w/32 \rceil$ and a an array (or equivalent) of length $r \cdot k$, invokes `q.generate(`$a+0$`, `$a+r \cdot k$`)` and then, iteratively for $i = -r, \dots, -1$, sets X_i to $\left(\sum_{j=0}^{k-1} a_{k(i+r)+j} \cdot 2^{32j}\right) \bmod m$. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

26.6.4 Random number engine adaptor class templates [rand.adapt]

26.6.4.1 In general [rand.adapt.general]

1 Each type instantiated from a class template specified in this subclause 26.6.4 meets the requirements of a random number engine adaptor (26.6.2.5) type.

2 Except where specified otherwise, the complexity of each function specified in this subclause 26.6.4 is constant.

3 Except where specified otherwise, no function described in this subclause 26.6.4 throws an exception.

4 Every function described in this subclause 26.6.4 that has a function parameter `q` of type `Sseq&` for a template type parameter named `Sseq` that is different from type `seed_seq` throws what and when the invocation of `q.generate` throws.

5 Descriptions are provided in this subclause 26.6.4 only for adaptor operations that are not described in subclause 26.6.2.5 or for operations where there is additional semantic information. In particular, declarations

for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

6 Each template specified in this subclause 26.6.4 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

26.6.4.2 Class template `discard_block_engine` [rand.adapt.disc]

1 A `discard_block_engine` random number engine adaptor produces random numbers selected from those produced by some base engine e. The state $\mathtt{x}_i$ of a `discard_block_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ of its base engine `e` and an additional integer n. The size of the state is the size of e's state plus 1.

2 The transition algorithm discards all but $r > 0$ values from each block of $p \geq r$ values delivered by e. The state transition is performed as follows: If $n \geq r$, advance the state of `e` from $\mathtt{e}_i$ to $\mathtt{e}_{i+p-r}$ and set n to 0. In any case, then increment n and advance `e`'s then-current state $\mathtt{e}_j$ to $\mathtt{e}_{j+1}$.

3 The generation algorithm yields the value returned by the last invocation of `e()` while advancing `e`'s state as described above.

```
template<class Engine, size_t p, size_t r>
  class discard_block_engine {
  public:
    // types
    using result_type = typename Engine::result_type;

    // engine characteristics
    static constexpr size_t block_size = p;
    static constexpr size_t used_block = r;
    static constexpr result_type min() { return Engine::min(); }
    static constexpr result_type max() { return Engine::max(); }

    // constructors and seeding functions
    discard_block_engine();
    explicit discard_block_engine(const Engine& e);
    explicit discard_block_engine(Engine&& e);
    explicit discard_block_engine(result_type s);
    template<class Sseq> explicit discard_block_engine(Sseq& q);
    void seed();
    void seed(result_type s);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);

    // property functions
    const Engine& base() const noexcept { return e; };

  private:
    Engine e;    // exposition only
    int n;       // exposition only
  };
```

4 The following relations shall hold: `0 < r` and `r <= p`.

5 The textual representation consists of the textual representation of `e` followed by the value of `n`.

6 In addition to its behavior pursuant to subclause 26.6.2.5, each constructor that is not a copy constructor sets `n` to 0.

26.6.4.3 Class template `independent_bits_engine` [rand.adapt.ibits]

1 An `independent_bits_engine` random number engine adaptor combines random numbers that are produced by some base engine e, so as to produce random numbers with a specified number of bits w. The state $\mathtt{x}_i$ of an `independent_bits_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ of its base engine `e`; the size of the state is the size of e's state.

2 The transition and generation algorithms are described in terms of the following integral constants:

(2.1) — Let $R = $ `e.max() - e.min() + 1` and $m = \lfloor \log_2 R \rfloor$.

(2.2) — With n as determined below, let $w_0 = \lfloor w/n \rfloor$, $n_0 = n - w \bmod n$, $y_0 = 2^{w_0} \lfloor R/2^{w_0} \rfloor$, and $y_1 = 2^{w_0+1} \lfloor R/2^{w_0+1} \rfloor$.

(2.3) — Let $n = \lceil w/m \rceil$ if and only if the relation $R - y_0 \leq \lfloor y_0/n \rfloor$ holds as a result. Otherwise let $n = 1 + \lceil w/m \rceil$.

[*Note*: The relation $w = n_0 w_0 + (n - n_0)(w_0 + 1)$ always holds. — *end note*]

3 The transition algorithm is carried out by invoking `e()` as often as needed to obtain n_0 values less than $y_0 +$ `e.min()` and $n - n_0$ values less than $y_1 +$ `e.min()`.

4 The generation algorithm uses the values produced while advancing the state as described above to yield a quantity S obtained as if by the following algorithm:

```
S = 0;
for (k = 0; k ≠ n₀; k += 1) {
 do u = e() - e.min(); while (u ≥ y₀);
 S = 2^w₀ · S + u mod 2^w₀;
}
for (k = n₀; k ≠ n; k += 1) {
 do u = e() - e.min(); while (u ≥ y₁);
 S = 2^(w₀+1) · S + u mod 2^(w₀+1);
}
```

```
template<class Engine, size_t w, class UIntType>
  class independent_bits_engine {
  public:
    // types
    using result_type = UIntType;

    // engine characteristics
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return 2^w - 1; }

    // constructors and seeding functions
    independent_bits_engine();
    explicit independent_bits_engine(const Engine& e);
    explicit independent_bits_engine(Engine&& e);
    explicit independent_bits_engine(result_type s);
    template<class Sseq> explicit independent_bits_engine(Sseq& q);
    void seed();
    void seed(result_type s);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);

    // property functions
    const Engine& base() const noexcept { return e; };

  private:
    Engine e;    // exposition only
  };
```

5 The following relations shall hold: `0 < w` and `w <= numeric_limits<result_type>::digits`.

6 The textual representation consists of the textual representation of `e`.

26.6.4.4 Class template `shuffle_order_engine` [rand.adapt.shuf]

1 A `shuffle_order_engine` random number engine adaptor produces the same random numbers that are produced by some base engine e, but delivers them in a different sequence. The state $\mathtt{x}_i$ of a `shuffle_order_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ of its base engine `e`, an additional value Y of the type delivered by `e`, and an additional sequence V of k values also of the type delivered by `e`. The size of the state is the size of e's state plus $k + 1$.

2 The transition algorithm permutes the values produced by e. The state transition is performed as follows:

(2.1) — Calculate an integer $j = \left\lfloor \frac{k\cdot(Y-e_{\min})}{e_{\max}-e_{\min}+1} \right\rfloor$.

(2.2) — Set Y to V_j and then set V_j to `e()`.

3 The generation algorithm yields the last value of `Y` produced while advancing `e`'s state as described above.

```
template<class Engine, size_t k>
  class shuffle_order_engine {
  public:
    // types
    using result_type = typename Engine::result_type;

    // engine characteristics
    static constexpr size_t table_size = k;
    static constexpr result_type min() { return Engine::min(); }
    static constexpr result_type max() { return Engine::max(); }

    // constructors and seeding functions
    shuffle_order_engine();
    explicit shuffle_order_engine(const Engine& e);
    explicit shuffle_order_engine(Engine&& e);
    explicit shuffle_order_engine(result_type s);
    template<class Sseq> explicit shuffle_order_engine(Sseq& q);
    void seed();
    void seed(result_type s);
    template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);

    // property functions
    const Engine& base() const noexcept { return e; };

  private:
    Engine e;           // exposition only
    result_type V[k];   // exposition only
    result_type Y;      // exposition only
  };
```

4 The following relation shall hold: `0 < k`.

5 The textual representation consists of the textual representation of `e`, followed by the `k` values of V, followed by the value of Y.

6 In addition to its behavior pursuant to subclause 26.6.2.5, each constructor that is not a copy constructor initializes `V[0]`, ..., `V[k-1]` and Y, in that order, with values returned by successive invocations of `e()`.

26.6.5 Engines and engine adaptors with predefined parameters [rand.predef]

```
using minstd_rand0 =
      linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647>;
```

1 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `minstd_rand0` ~~shall produce~~produces the value 1043618065.

```
using minstd_rand =
      linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647>;
```

2 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `minstd_rand` ~~shall produce~~produces the value 399268537.

```
using mt19937 =
      mersenne_twister_engine<uint_fast32_t,
```

```
   32,624,397,31,0x9908b0df,11,0xffffffff,7,0x9d2c5680,15,0xefc60000,18,1812433253>;
```

3 *Required behavior:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `mt19937` ~~shall produce~~produces the value 4123659995.

```
using mt19937_64 =
      mersenne_twister_engine<uint_fast64_t,
       64,312,156,31,0xb5026f5aa96619e9,29,
       0x5555555555555555,17,
       0x71d67fffeda60000,37,
       0xfff7eee000000000,43,
       6364136223846793005>;
```

4 *Required behavior:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `mt19937_64` ~~shall produce~~produces the value 9981545732273789042.

```
using ranlux24_base =
      subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>;
```

5 *Required behavior:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `ranlux24_base` ~~shall produce~~produces the value 7937952.

```
using ranlux48_base =
      subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>;
```

6 *Required behavior:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `ranlux48_base` ~~shall produce~~produces the value 61839128582725.

```
using ranlux24 = discard_block_engine<ranlux24_base, 223, 23>;
```

7 *Required behavior:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `ranlux24` ~~shall produce~~produces the value 9901578.

```
using ranlux48 = discard_block_engine<ranlux48_base, 389, 11>;
```

8 *Required behavior:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `ranlux48` ~~shall produce~~produces the value 249142670248501.

```
using knuth_b = shuffle_order_engine<minstd_rand0,256>;
```

9 *Required behavior:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `knuth_b` ~~shall produce~~produces the value 1112339016.

```
using default_random_engine = implementation-defined;
```

10 *Remarks:* The choice of engine type named by this `typedef` is implementation-defined. [*Note*: The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use. Because different implementations may select different underlying engine types, code that uses this `typedef` need not generate identical sequences across implementations. — *end note*]

26.6.6 Class `random_device` [rand.device]

1 A `random_device` uniform random bit generator produces nondeterministic random numbers.

2 If implementation limitations prevent generating nondeterministic random numbers, the implementation may employ a random number engine.

```
class random_device {
public:
  // types
  using result_type = unsigned int;

  // generator characteristics
  static constexpr result_type min() { return numeric_limits<result_type>::min(); }
  static constexpr result_type max() { return numeric_limits<result_type>::max(); }

  // constructors
  random_device() : random_device(implementation-defined) {}
  explicit random_device(const string& token);
```

```
    // generating functions
    result_type operator()();

    // property functions
    double entropy() const noexcept;

    // no copy functions
    random_device(const random_device&) = delete;
    void operator=(const random_device&) = delete;
  };
```

`explicit random_device(const string& token);`

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `random_device` nondeterministic uniform random bit generator object.~~ The semantics of the `token` parameter and the token value used by the default constructor are implementation-defined.[245]

4 *Throws:* A value of an implementation-defined type derived from `exception` if the `random_device` could not be initialized.

`double entropy() const noexcept;`

5 *Returns:* If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate[246] for the random numbers returned by `operator()`, in the range `min()` to $\log_2(\texttt{max()} + 1)$.

`result_type operator()();`

6 *Returns:* A nondeterministic random value, uniformly distributed between `min()` and `max()` (inclusive). It is implementation-defined how these values are generated.

7 *Throws:* A value of an implementation-defined type derived from `exception` if a random number could not be obtained.

26.6.7 Utilities [rand.util]

26.6.7.1 Class seed_seq [rand.util.seedseq]

```
  class seed_seq {
  public:
    // types
    using result_type = uint_least32_t;

    // constructors
    seed_seq();
    template<class T>
      seed_seq(initializer_list<T> il);
    template<class InputIterator>
      seed_seq(InputIterator begin, InputIterator end);

    // generating functions
    template<class RandomAccessIterator>
      void generate(RandomAccessIterator begin, RandomAccessIterator end);

    // property functions
    size_t size() const noexcept;
    template<class OutputIterator>
      void param(OutputIterator dest) const;

    // no copy functions
    seed_seq(const seed_seq&) = delete;
    void operator=(const seed_seq&) = delete;
```

245) The parameter is intended to allow an implementation to differentiate between different sources of randomness.
246) If a device has n states whose respective probabilities are $P_0, \ldots, P_{n-1}$, the device entropy S is defined as $S = -\sum_{i=0}^{n-1} P_i \cdot \log P_i$.

```
private:
  vector<result_type> v;    // exposition only
};
```

```
seed_seq();
```

1 *Effects:* Constructs a `seed_seq` object as if by default-constructing its member `v`.

2 *Ensures:* `v.empty()` is true.

3 *Throws:* Nothing.

```
template<class T>
 seed_seq(initializer_list<T> il);
```

4 ~~*Requires:*~~ *Mandates:* `T` ~~shall be~~is an integer type.

5 *Effects:* Same as `seed_seq(il.begin(), il.end())`.

```
template<class InputIterator>
  seed_seq(InputIterator begin, InputIterator end);
```

6 *Mandates:* `iterator_traits<InputIterator>::value_type` is an integer type.

7 ~~*Requires:*~~ *Expects:* `InputIterator` ~~shall meet~~meets the *Cpp17InputIterator* requirements (??). Moreover, `iterator_traits<InputIterator>::value_type` shall denote an integer type.

8 *Effects:* ~~Constructs a `seed_seq` object~~Initializes `v` by the following algorithm:

```
for (InputIterator s = begin; s != end; ++s)
 v.push_back((*s) mod 2^32);
```

```
template<class RandomAccessIterator>
  void generate(RandomAccessIterator begin, RandomAccessIterator end);
```

9 *Mandates:* `iterator_traits<RandomAccessIterator>::value_type` is an unsigned integer type capable of accommodating 32-bit quantities.

10 ~~*Requires:*~~ *Expects:* `RandomAccessIterator` ~~shall meet~~meets the *Cpp17RandomAccessIterator* requirements (??) and the requirements of a mutable iterator. Moreover, `iterator_traits<RandomAccessIterator>::value_type` shall denote an unsigned integer type capable of accommodating 32-bit quantities.

11 *Effects:* Does nothing if `begin == end`. Otherwise, with $s = \texttt{v.size()}$ and $n = \texttt{end} - \texttt{begin}$, fills the supplied range $[\texttt{begin}, \texttt{end})$ according to the following algorithm in which each operation is to be carried out modulo 2^{32}, each indexing operator applied to `begin` is to be taken modulo n, and $T(x)$ is defined as $x \operatorname{xor} (x \operatorname{rshift} 27)$:

(11.1) — By way of initialization, set each element of the range to the value `0x8b8b8b8b`. Additionally, for use in subsequent steps, let $p = (n - t)/2$ and let $q = p + t$, where

$$t = (n \geq 623)\ ?\ 11 : (n \geq 68)\ ?\ 7 : (n \geq 39)\ ?\ 5 : (n \geq 7)\ ?\ 3 : (n - 1)/2;$$

(11.2) — With m as the larger of $s + 1$ and n, transform the elements of the range: iteratively for $k = 0, \ldots, m - 1$, calculate values

$$\begin{aligned} r_1 &= 1664525 \cdot T(\texttt{begin}[k] \operatorname{xor} \texttt{begin}[k+p] \operatorname{xor} \texttt{begin}[k-1]) \\ r_2 &= r_1 + \begin{cases} s & , k = 0 \\ k \bmod n + \texttt{v}[k-1] & , 0 < k \leq s \\ k \bmod n & , s < k \end{cases} \end{aligned}$$

and, in order, increment $\texttt{begin}[k+p]$ by r_1, increment $\texttt{begin}[k+q]$ by r_2, and set $\texttt{begin}[k]$ to r_2.

(11.3) — Transform the elements of the range again, beginning where the previous step ended: iteratively for $k = m, \ldots, m + n - 1$, calculate values

$$\begin{aligned} r_3 &= 1566083941 \cdot T(\texttt{begin}[k] + \texttt{begin}[k+p] + \texttt{begin}[k-1]) \\ r_4 &= r_3 - (k \bmod n) \end{aligned}$$

and, in order, update $\texttt{begin}[k+p]$ by xoring it with r_3, update $\texttt{begin}[k+q]$ by xoring it with r_4, and set $\texttt{begin}[k]$ to r_4.

12 *Throws:* What and when `RandomAccessIterator` operations of `begin` and `end` throw.

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

13 *Returns:* The number of 32-bit units that would be returned by a call to `param()`.

14 *Complexity:* Constant time.

```
template<class OutputIterator>
  void param(OutputIterator dest) const;
```

15 *Mandates:* Values of type `result_type` are writable to `dest`.

16 ~~*Requires:*~~ *Expects:* `OutputIterator` ~~shall meet~~meets the *Cpp17OutputIterator* requirements (??). Moreover, the expression `*dest = rt` shall be valid for a value `rt` of type `result_type`.

17 *Effects:* Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:

```
copy(v.begin(), v.end(), dest);
```

18 *Throws:* What and when `OutputIterator` operations of `dest` throw.

26.6.7.2 Function template `generate_canonical` [rand.util.canonical]

1 Each function instantiated from the template described in this subclause 26.6.7.2 maps the result of one or more invocations of a supplied uniform random bit generator `g` to one member of the specified `RealType` such that, if the values g_i produced by `g` are uniformly distributed, the instantiation's results t_j, $0 \le t_j < 1$, are distributed as uniformly as possible as specified below.

2 [*Note*: Obtaining a value in this way can be a useful step in the process of transforming a value generated by a uniform random bit generator into a value that can be delivered by a random number distribution. — *end note*]

```
template<class RealType, size_t bits, class URBG>
 RealType generate_canonical(URBG& g);
```

3 *Complexity:* Exactly $k = \max(1, \lceil b / \log_2 R \rceil)$ invocations of `g`, where b[247] is the lesser of `numeric_limits<RealType>::digits` and `bits`, and R is the value of $\texttt{g.max()} - \texttt{g.min()} + 1$.

4 *Effects:* Invokes `g()` k times to obtain values $g_0, \ldots, g_{k-1}$, respectively. Calculates a quantity

$$S = \sum_{i=0}^{k-1} (g_i - \texttt{g.min()}) \cdot R^i$$

using arithmetic of type `RealType`.

5 *Returns:* S/R^k.

6 *Throws:* What and when `g` throws.

26.6.8 Random number distribution class templates [rand.dist]

26.6.8.1 In general [rand.dist.general]

1 Each type instantiated from a class template specified in this subclause 26.6.8 meets the requirements of a random number distribution (26.6.2.6) type.

2 Descriptions are provided in this subclause 26.6.8 only for distribution operations that are not described in 26.6.2.6 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

3 The algorithms for producing each of the specified distributions are implementation-defined.

4 The value of each probability density function $p(z)$ and of each discrete probability function $P(z_i)$ specified in this subclause is 0 everywhere outside its stated domain.

247) b is introduced to avoid any attempt to produce more bits of randomness than can be held in `RealType`.

26.6.8.2 Uniform distributions [rand.dist.uni]

26.6.8.2.1 Class template `uniform_int_distribution` [rand.dist.uni.int]

1 A `uniform_int_distribution` random number distribution produces random integers i, $a \le i \le b$, distributed according to the constant discrete probability function

$$P(i \,|\, a, b) = 1/(b - a + 1) \;.$$

```
template<class IntType = int>
  class uniform_int_distribution {
  public:
    // types
    using result_type = IntType;
    using param_type  = unspecified;

    // constructors and reset functions
    uniform_int_distribution() : uniform_int_distribution(0) {}
    explicit uniform_int_distribution(IntType a, IntType b = numeric_limits<IntType>::max());
    explicit uniform_int_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

`explicit uniform_int_distribution(IntType a, IntType b = numeric_limits<IntType>::max());`

2 ~~*Requires:*~~ *Expects:* `a` $\le$ `b`.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `uniform_int_distribution` object;~~ `a` and `b` correspond to the respective parameters of the distribution.

`result_type a() const;`

4 *Returns:* The value of the `a` parameter with which the object was constructed.

`result_type b() const;`

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.6.8.2.2 Class template `uniform_real_distribution` [rand.dist.uni.real]

1 A `uniform_real_distribution` random number distribution produces random numbers x, $a \le x < b$, distributed according to the constant probability density function

$$p(x \,|\, a, b) = 1/(b - a) \;.$$

[*Note*: This implies that $p(x \,|\, a, b)$ is undefined when `a == b`. — *end note*]

```
template<class RealType = double>
  class uniform_real_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;
```

```
    // constructors and reset functions
    uniform_real_distribution() : uniform_real_distribution(0.0) {}
    explicit uniform_real_distribution(RealType a, RealType b = 1.0);
    explicit uniform_real_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

`explicit uniform_real_distribution(RealType a, RealType b = 1.0);`

2 ~~*Requires:*~~ *Expects:* $a \leq b$ and $b - a \leq$ `numeric_limits<RealType>::max()`.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `uniform_real_distribution` object;~~ `a` and `b` correspond to the respective parameters of the distribution.

`result_type a() const;`

4 *Returns:* The value of the `a` parameter with which the object was constructed.

`result_type b() const;`

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.6.8.3 Bernoulli distributions [rand.dist.bern]

26.6.8.3.1 Class `bernoulli_distribution` [rand.dist.bern.bernoulli]

1 A `bernoulli_distribution` random number distribution produces `bool` values b distributed according to the discrete probability function

$$P(b\,|\,p) = \begin{cases} p & \text{if } b = \texttt{true}, \text{ or} \\ 1-p & \text{if } b = \texttt{false}. \end{cases}$$

```
class bernoulli_distribution {
public:
  // types
  using result_type = bool;
  using param_type  = unspecified;

  // constructors and reset functions
  bernoulli_distribution() : bernoulli_distribution(0.5) {}
  explicit bernoulli_distribution(double p);
  explicit bernoulli_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URBG>
    result_type operator()(URBG& g);
  template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

  // property functions
  double p() const;
  param_type param() const;
```

```
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

`explicit bernoulli_distribution(double p);`

2 ~~*Requires:*~~ *Expects:* $0 \leq$ p ≤ 1.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `bernoulli_distribution` object;~~ p corresponds to the parameter of the distribution.

`double p() const;`

4 *Returns:* The value of the p parameter with which the object was constructed.

26.6.8.3.2 Class template `binomial_distribution` [rand.dist.bern.bin]

1 A `binomial_distribution` random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i \mid t, p) = \binom{t}{i} \cdot p^i \cdot (1-p)^{t-i} \ .$$

```
  template<class IntType = int>
    class binomial_distribution {
    public:
      // types
      using result_type = IntType;
      using param_type  = unspecified;

      // constructors and reset functions
      binomial_distribution() : binomial_distribution(1) {}
      explicit binomial_distribution(IntType t, double p = 0.5);
      explicit binomial_distribution(const param_type& parm);
      void reset();

      // generating functions
      template<class URBG>
        result_type operator()(URBG& g);
      template<class URBG>
        result_type operator()(URBG& g, const param_type& parm);

      // property functions
      IntType t() const;
      double p() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
      result_type max() const;
    };
```

`explicit binomial_distribution(IntType t, double p = 0.5);`

2 ~~*Requires:*~~ *Expects:* $0 \leq$ p ≤ 1 and $0 \leq$ t.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `binomial_distribution` object;~~ t and p correspond to the respective parameters of the distribution.

`IntType t() const;`

4 *Returns:* The value of the t parameter with which the object was constructed.

`double p() const;`

5 *Returns:* The value of the p parameter with which the object was constructed.

26.6.8.3.3 Class template `geometric_distribution` [rand.dist.bern.geo]

1 A `geometric_distribution` random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i \,|\, p) = p \cdot (1-p)^i \; .$$

```
template<class IntType = int>
  class geometric_distribution {
  public:
    // types
    using result_type = IntType;
    using param_type  = unspecified;

    // constructors and reset functions
    geometric_distribution() : geometric_distribution(0.5) {}
    explicit geometric_distribution(double p);
    explicit geometric_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

`explicit geometric_distribution(double p);`

2 ~~*Requires:*~~ *Expects:* $0 < \texttt{p} < 1$.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `geometric_distribution` object;~~ `p` corresponds to the parameter of the distribution.

`double p() const;`

4 *Returns:* The value of the `p` parameter with which the object was constructed.

26.6.8.3.4 Class template `negative_binomial_distribution` [rand.dist.bern.negbin]

1 A `negative_binomial_distribution` random number distribution produces random integers $i \geq 0$ distributed according to the discrete probability function

$$P(i \,|\, k, p) = \binom{k+i-1}{i} \cdot p^k \cdot (1-p)^i \; .$$

[*Note*: This implies that $P(i \,|\, k, p)$ is undefined when `p == 1`. — *end note*]

```
template<class IntType = int>
  class negative_binomial_distribution {
  public:
    // types
    using result_type = IntType;
    using param_type  = unspecified;

    // constructor and reset functions
    negative_binomial_distribution() : negative_binomial_distribution(1) {}
    explicit negative_binomial_distribution(IntType k, double p = 0.5);
    explicit negative_binomial_distribution(const param_type& parm);
    void reset();
```

```
    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    IntType k() const;
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit negative_binomial_distribution(IntType k, double p = 0.5);
```

2 ~~*Requires:*~~ *Expects:* $0 < \texttt{p} \leq 1$ and $0 < \texttt{k}$.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `negative_binomial_distribution` object;~~ `k` and `p` correspond to the respective parameters of the distribution.

```
IntType k() const;
```

4 *Returns:* The value of the `k` parameter with which the object was constructed.

```
double p() const;
```

5 *Returns:* The value of the `p` parameter with which the object was constructed.

26.6.8.4 Poisson distributions [rand.dist.pois]

26.6.8.4.1 Class template `poisson_distribution` [rand.dist.pois.poisson]

1 A `poisson_distribution` random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i\,|\,\mu) = \frac{e^{-\mu}\mu^i}{i\,!}\ .$$

The distribution parameter μ is also known as this distribution's *mean*.

```
  template<class IntType = int>
    class poisson_distribution
    {
    public:
      // types
      using result_type = IntType;
      using param_type  = unspecified;

      // constructors and reset functions
      poisson_distribution() : poisson_distribution(1.0) {}
      explicit poisson_distribution(double mean);
      explicit poisson_distribution(const param_type& parm);
      void reset();

      // generating functions
      template<class URBG>
        result_type operator()(URBG& g);
      template<class URBG>
        result_type operator()(URBG& g, const param_type& parm);

      // property functions
      double mean() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
      result_type max() const;
    };
```

```
explicit poisson_distribution(double mean);
```

2 ~~*Requires:*~~ *Expects:* $0 <$ `mean`.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `poisson_distribution` object;~~ `mean` corresponds to the parameter of the distribution.

```
double mean() const;
```

4 *Returns:* The value of the `mean` parameter with which the object was constructed.

26.6.8.4.2 Class template `exponential_distribution` [rand.dist.pois.exp]

1 An `exponential_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \,|\, \lambda) = \lambda e^{-\lambda x} \ .$$

```
template<class RealType = double>
  class exponential_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructors and reset functions
    exponential_distribution() : exponential_distribution(1.0) {}
    explicit exponential_distribution(RealType lambda);
    explicit exponential_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType lambda() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit exponential_distribution(RealType lambda);
```

2 ~~*Requires:*~~ *Expects:* $0 <$ `lambda`.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs an `exponential_distribution` object;~~ `lambda` corresponds to the parameter of the distribution.

```
RealType lambda() const;
```

4 *Returns:* The value of the `lambda` parameter with which the object was constructed.

26.6.8.4.3 Class template `gamma_distribution` [rand.dist.pois.gamma]

1 A `gamma_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \,|\, \alpha, \beta) = \frac{e^{-x/\beta}}{\beta^{\alpha} \cdot \Gamma(\alpha)} \cdot x^{\,\alpha-1} \ .$$

```
template<class RealType = double>
  class gamma_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;
```

```
    // constructors and reset functions
    gamma_distribution() : gamma_distribution(1.0) {}
    explicit gamma_distribution(RealType alpha, RealType beta = 1.0);
    explicit gamma_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType alpha() const;
    RealType beta() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit gamma_distribution(RealType alpha, RealType beta = 1.0);
```

2 ~~*Requires:*~~ *Expects:* $0 <$ `alpha` and $0 <$ `beta`.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `gamma_distribution` object;~~ `alpha` and `beta` correspond to the parameters of the distribution.

```
RealType alpha() const;
```

4 *Returns:* The value of the `alpha` parameter with which the object was constructed.

```
RealType beta() const;
```

5 *Returns:* The value of the `beta` parameter with which the object was constructed.

26.6.8.4.4 Class template `weibull_distribution` [rand.dist.pois.weibull]

1 A `weibull_distribution` random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$p(x \,|\, a, b) = \frac{a}{b} \cdot \left(\frac{x}{b}\right)^{a-1} \cdot \exp\left(-\left(\frac{x}{b}\right)^{a}\right) .$$

```
  template<class RealType = double>
    class weibull_distribution {
    public:
      // types
      using result_type = RealType;
      using param_type  = unspecified;

      // constructor and reset functions
      weibull_distribution() : weibull_distribution(1.0) {}
      explicit weibull_distribution(RealType a, RealType b = 1.0);
      explicit weibull_distribution(const param_type& parm);
      void reset();

      // generating functions
      template<class URBG>
        result_type operator()(URBG& g);
      template<class URBG>
        result_type operator()(URBG& g, const param_type& parm);

      // property functions
      RealType a() const;
      RealType b() const;
      param_type param() const;
```

```
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit weibull_distribution(RealType a, RealType b = 1.0);
```

2 ~~*Requires:*~~ *Expects:* $0 < \texttt{a}$ and $0 < \texttt{b}$.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `weibull_distribution` object;~~ `a` and `b` correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

```
RealType b() const;
```

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.6.8.4.5 Class template `extreme_value_distribution` [rand.dist.pois.extreme]

1 An `extreme_value_distribution` random number distribution produces random numbers x distributed according to the probability density function[248]

$$p(x \,|\, a, b) = \frac{1}{b} \cdot \exp\left(\frac{a-x}{b} - \exp\left(\frac{a-x}{b}\right)\right) .$$

```
template<class RealType = double>
  class extreme_value_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    extreme_value_distribution() : extreme_value_distribution(0.0) {}
    explicit extreme_value_distribution(RealType a, RealType b = 1.0);
    explicit extreme_value_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit extreme_value_distribution(RealType a, RealType b = 1.0);
```

2 ~~*Requires:*~~ *Expects:* $0 < \texttt{b}$.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs an `extreme_value_distribution` object;~~ `a` and `b` correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

248) The distribution corresponding to this probability density function is also known (with a possible change of variable) as the Gumbel Type I, the log-Weibull, or the Fisher-Tippett Type I distribution.

```
RealType b() const;
```

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.6.8.5 Normal distributions [rand.dist.norm]

26.6.8.5.1 Class template `normal_distribution` [rand.dist.norm.normal]

1 A `normal_distribution` random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

The distribution parameters μ and σ are also known as this distribution's *mean* and *standard deviation*.

```
template<class RealType = double>
  class normal_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructors and reset functions
    normal_distribution() : normal_distribution(0.0) {}
    explicit normal_distribution(RealType mean, RealType stddev = 1.0);
    explicit normal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType mean() const;
    RealType stddev() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit normal_distribution(RealType mean, RealType stddev = 1.0);
```

2 ~~*Requires:*~~ *Expects:* $0 <$ `stddev`.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `normal_distribution` object;~~ `mean` and `stddev` correspond to the respective parameters of the distribution.

```
RealType mean() const;
```

4 *Returns:* The value of the `mean` parameter with which the object was constructed.

```
RealType stddev() const;
```

5 *Returns:* The value of the `stddev` parameter with which the object was constructed.

26.6.8.5.2 Class template `lognormal_distribution` [rand.dist.norm.lognormal]

1 A `lognormal_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid m, s) = \frac{1}{sx\sqrt{2\pi}} \cdot \exp\left(-\frac{(\ln x - m)^2}{2s^2}\right).$$

```
template<class RealType = double>
  class lognormal_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    lognormal_distribution() : lognormal_distribution(0.0) {}
    explicit lognormal_distribution(RealType m, RealType s = 1.0);
    explicit lognormal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType m() const;
    RealType s() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

`explicit lognormal_distribution(RealType m, RealType s = 1.0);`

2 ~~*Requires:*~~ *Expects:* $0 < \mathtt{s}$.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `lognormal_distribution` object;~~ `m` and `s` correspond to the respective parameters of the distribution.

`RealType m() const;`

4 *Returns:* The value of the `m` parameter with which the object was constructed.

`RealType s() const;`

5 *Returns:* The value of the `s` parameter with which the object was constructed.

26.6.8.5.3 Class template `chi_squared_distribution` [rand.dist.norm.chisq]

1 A `chi_squared_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x\,|\,n) = \frac{x^{(n/2)-1} \cdot e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}} \ .$$

```
template<class RealType = double>
  class chi_squared_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    chi_squared_distribution() : chi_squared_distribution(1) {}
    explicit chi_squared_distribution(RealType n);
    explicit chi_squared_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
```

```
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit chi_squared_distribution(RealType n);
```

2 ~~*Requires:*~~ *Expects:* $0 < \texttt{n}$.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a chi_squared_distribution object;~~ `n` corresponds to the parameter of the distribution.

```
RealType n() const;
```

4 *Returns:* The value of the `n` parameter with which the object was constructed.

26.6.8.5.4 Class template `cauchy_distribution` [rand.dist.norm.cauchy]

1 A `cauchy_distribution` random number distribution produces random numbers x distributed according to the probability density function

$$p(x\,|\,a,b) = \left(\pi b \left(1 + \left(\frac{x-a}{b}\right)^2\right)\right)^{-1} .$$

```
template<class RealType = double>
  class cauchy_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    cauchy_distribution() : cauchy_distribution(0.0) {}
    explicit cauchy_distribution(RealType a, RealType b = 1.0);
    explicit cauchy_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit cauchy_distribution(RealType a, RealType b = 1.0);
```

2 ~~*Requires:*~~ *Expects:* $0 < \texttt{b}$.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a cauchy_distribution object;~~ `a` and `b` correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

```
RealType b() const;
```

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.6.8.5.5 Class template `fisher_f_distribution` [rand.dist.norm.f]

1 A `fisher_f_distribution` random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$p(x \mid m, n) = \frac{\Gamma\big((m+n)/2\big)}{\Gamma(m/2)\,\Gamma(n/2)} \cdot \left(\frac{m}{n}\right)^{m/2} \cdot x^{(m/2)-1} \cdot \left(1 + \frac{mx}{n}\right)^{-(m+n)/2} .$$

```
template<class RealType = double>
  class fisher_f_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    fisher_f_distribution() : fisher_f_distribution(1) {}
    explicit fisher_f_distribution(RealType m, RealType n = 1);
    explicit fisher_f_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType m() const;
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit fisher_f_distribution(RealType m, RealType n = 1);
```

2 ~~*Requires:*~~ *Expects:* $0 <$ `m` and $0 <$ `n`.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `fisher_f_distribution` object;~~ `m` and `n` correspond to the respective parameters of the distribution.

```
RealType m() const;
```

4 *Returns:* The value of the `m` parameter with which the object was constructed.

```
RealType n() const;
```

5 *Returns:* The value of the `n` parameter with which the object was constructed.

26.6.8.5.6 Class template `student_t_distribution` [rand.dist.norm.t]

1 A `student_t_distribution` random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid n) = \frac{1}{\sqrt{n\pi}} \cdot \frac{\Gamma\big((n+1)/2\big)}{\Gamma(n/2)} \cdot \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2} .$$

```
template<class RealType = double>
  class student_t_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;
```

```
    // constructor and reset functions
    student_t_distribution() : student_t_distribution(1) {}
    explicit student_t_distribution(RealType n);
    explicit student_t_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

`explicit student_t_distribution(RealType n);`

2 ~~*Requires:*~~ *Expects:* $0 < n$.

3 ~~*Effects:*~~ *Remarks:* ~~Constructs a `student_t_distribution` object;~~ `n` corresponds to the parameter of the distribution.

`RealType n() const;`

4 *Returns:* The value of the `n` parameter with which the object was constructed.

26.6.8.6 Sampling distributions [rand.dist.samp]

26.6.8.6.1 Class template `discrete_distribution` [rand.dist.samp.discrete]

1 A `discrete_distribution` random number distribution produces random integers i, $0 \leq i < n$, distributed according to the discrete probability function

$$P(i \,|\, p_0, \dots, p_{n-1}) = p_i \ .$$

2 Unless specified otherwise, the distribution parameters are calculated as: $p_k = w_k/S$ for $k = 0, \dots, n-1$, in which the values w_k, commonly known as the *weights*, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: $0 < S = w_0 + \cdots + w_{n-1}$.

```
template<class IntType = int>
  class discrete_distribution {
  public:
    // types
    using result_type = IntType;
    using param_type  = unspecified;

    // constructor and reset functions
    discrete_distribution();
    template<class InputIterator>
      discrete_distribution(InputIterator firstW, InputIterator lastW);
    discrete_distribution(initializer_list<double> wl);
    template<class UnaryOperation>
      discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);
    explicit discrete_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);
```

```
    // property functions
    vector<double> probabilities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
discrete_distribution();
```

3 *Effects:* Constructs a `discrete_distribution` object with $n = 1$ and $p_0 = 1$. [*Note*: Such an object will always deliver the value 0. — *end note*]

```
template<class InputIterator>
  discrete_distribution(InputIterator firstW, InputIterator lastW);
```

4 *Mandates:* `is_convertible_v<iterator_traits<InputIterator>::value_type, double>` is true.

5 ~~*Requires:*~~ *Expects:* `InputIterator` ~~shall meet~~meets the *Cpp17InputIterator* requirements (??). ~~Moreover, `iterator_traits<InputIterator>::value_type` shall denote a type that is convertible to `double`.~~ If `firstW == lastW`, let $n = 1$ and $w_0 = 1$. Otherwise, $[$`firstW`, `lastW`$)$ ~~shall form~~forms a sequence w of length $n > 0$.

6 *Effects:* Constructs a `discrete_distribution` object with probabilities given by the formula above.

```
discrete_distribution(initializer_list<double> wl);
```

7 *Effects:* Same as `discrete_distribution(wl.begin(), wl.end())`.

```
template<class UnaryOperation>
  discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);
```

8 *Mandates:* `is_invocable_r_v<double, UnaryOperation&, double>` is true.

9 ~~*Requires:*~~ *Expects:* ~~Each instance of type `UnaryOperation` shall be a function object (??) whose return type shall be convertible to `double`.~~ ~~Moreover, `double` shall be convertible to the type of `UnaryOperation`'s sole parameter.~~ If `nw` $= 0$, let $n = 1$, otherwise let $n =$ `nw`. The relation $0 < \delta = ($`xmax` $-$ `xmin`$)/n$ ~~shall hold~~holds.

10 *Effects:* Constructs a `discrete_distribution` object with probabilities given by the formula above, using the following values: If `nw` $= 0$, let $w_0 = 1$. Otherwise, let $w_k =$ `fw`$($`xmin` $+ k \cdot \delta + \delta/2)$ for $k = 0, \ldots, n-1$.

11 *Complexity:* The number of invocations of `fw` ~~shall~~does not exceed n.

```
vector<double> probabilities() const;
```

12 *Returns:* A `vector<double>` whose `size` member returns n and whose `operator[]` member returns p_k when invoked with argument k for $k = 0, \ldots, n-1$.

26.6.8.6.2 Class template `piecewise_constant_distribution` [rand.dist.samp.pconst]

1 A `piecewise_constant_distribution` random number distribution produces random numbers x, $b_0 \le x < b_n$, uniformly distributed over each subinterval $[b_i, b_{i+1})$ according to the probability density function

$$p(x \,|\, b_0, \ldots, b_n,\ \rho_0, \ldots, \rho_{n-1}) = \rho_i \text{ , for } b_i \le x < b_{i+1}.$$

2 The $n + 1$ distribution parameters b_i, also known as this distribution's *interval boundaries*, shall satisfy the relation $b_i < b_{i+1}$ for $i = 0, \ldots, n-1$. Unless specified otherwise, the remaining n distribution parameters are calculated as:

$$\rho_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \text{ for } k = 0, \ldots, n-1 \text{ ,}$$

in which the values w_k, commonly known as the *weights*, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: $0 < S = w_0 + \cdots + w_{n-1}$.

```
template<class RealType = double>
  class piecewise_constant_distribution {
  public:
    // types
    using result_type = RealType;
```

```
    using param_type  = unspecified;

    // constructor and reset functions
    piecewise_constant_distribution();
    template<class InputIteratorB, class InputIteratorW>
      piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                      InputIteratorW firstW);
    template<class UnaryOperation>
      piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);
    template<class UnaryOperation>
      piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax,
                                      UnaryOperation fw);
    explicit piecewise_constant_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    vector<result_type> intervals() const;
    vector<result_type> densities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

`piecewise_constant_distribution();`

3 *Effects:* Constructs a `piecewise_constant_distribution` object with $n = 1$, $\rho_0 = 1$, $b_0 = 0$, and $b_1 = 1$.

```
template<class InputIteratorB, class InputIteratorW>
 piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                 InputIteratorW firstW);
```

4 *Mandates:* Both of

(4.1) — `is_convertible_v<iterator_traits<InputIteratorB>::value_type, double>`

(4.2) — `is_convertible_v<iterator_traits<InputIteratorW>::value_type, double>`

are `true`.

5 ~~*Requires:*~~ *Expects:* `InputIteratorB` and `InputIteratorW` ~~shall~~ each meet the *Cpp17InputIterator* requirements (??). Moreover, the *id-expressions* `iterator_traits<InputIteratorB>::value_type` and `iterator_traits<InputIteratorW>::value_type` shall each denote a type that is convertible to `double`. If `firstB == lastB` or `++firstB == lastB`, let $n = 1$, $w_0 = 1$, $b_0 = 0$, and $b_1 = 1$. Otherwise, $[$`firstB`, `lastB`$)$ ~~shall form~~forms a sequence b of length $n+1$, the length of the sequence w starting from `firstW` ~~shall be~~is at least n, and any w_k for $k \geq n$ ~~shall be~~are ignored by the distribution.

6 *Effects:* Constructs a `piecewise_constant_distribution` object with parameters as specified above.

```
template<class UnaryOperation>
 piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);
```

7 *Mandates:* `is_invocable_r_v<double, UnaryOperation&, double>` is true.

8 *Requires:* Each instance of type `UnaryOperation` shall be a function object (??) whose return type shall be convertible to `double`. Moreover, `double` shall be convertible to the type of `UnaryOperation`'s sole parameter.

9 *Effects:* Constructs a `piecewise_constant_distribution` object with parameters taken or calculated from the following values: If `bl.size()` < 2, let $n = 1$, $w_0 = 1$, $b_0 = 0$, and $b_1 = 1$. Otherwise, let $[$`bl.begin()`, `bl.end()`$)$ form a sequence $b_0, \ldots, b_n$, and let $w_k = \texttt{fw}\big((b_{k+1} + b_k)/2\big)$ for $k = 0, \ldots, n-1$.

10 *Complexity:* The number of invocations of `fw` ~~shall~~does not exceed n.

```
template<class UnaryOperation>
 piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
```

11 *Mandates:* `is_invocable_r_v<double, UnaryOperation&, double>` is true.

12 ~~*Requires:*~~ *Expects:* ~~Each instance of type `UnaryOperation` shall be a function object (??) whose return type shall be convertible to `double`.~~ Moreover, `double` shall be convertible to the type of `UnaryOperation`'s sole parameter. If `nw` $= 0$, let $n = 1$, otherwise let $n =$ `nw`. The relation $0 < \delta = (\texttt{xmax} - \texttt{xmin})/n$ ~~shall hold~~holds.

13 *Effects:* Constructs a `piecewise_constant_distribution` object with parameters taken or calculated from the following values: Let $b_k = \texttt{xmin} + k \cdot \delta$ for $k = 0, \ldots, n$, and $w_k = \texttt{fw}(b_k + \delta/2)$ for $k = 0, \ldots, n-1$.

14 *Complexity:* The number of invocations of `fw` ~~shall~~does not exceed n.

```
vector<result_type> intervals() const;
```

15 *Returns:* A `vector<result_type>` whose `size` member returns $n + 1$ and whose `operator[]` member returns b_k when invoked with argument k for $k = 0, \ldots, n$.

```
vector<result_type> densities() const;
```

16 *Returns:* A `vector<result_type>` whose `size` member returns n and whose `operator[]` member returns ρ_k when invoked with argument k for $k = 0, \ldots, n-1$.

26.6.8.6.3 Class template `piecewise_linear_distribution` [rand.dist.samp.plinear]

1 A `piecewise_linear_distribution` random number distribution produces random numbers x, $b_0 \le x < b_n$, distributed over each subinterval $[b_i, b_{i+1})$ according to the probability density function

$$p(x \,|\, b_0, \ldots, b_n,\ \rho_0, \ldots, \rho_n) = \rho_i \cdot \frac{b_{i+1} - x}{b_{i+1} - b_i} + \rho_{i+1} \cdot \frac{x - b_i}{b_{i+1} - b_i}\ , \text{ for } b_i \le x < b_{i+1}.$$

2 The $n + 1$ distribution parameters b_i, also known as this distribution's *interval boundaries*, shall satisfy the relation $b_i < b_{i+1}$ for $i = 0, \ldots, n-1$. Unless specified otherwise, the remaining $n + 1$ distribution parameters are calculated as $\rho_k = w_k / S$ for $k = 0, \ldots, n$, in which the values w_k, commonly known as the *weights at boundaries*, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold:

$$0 < S = \frac{1}{2} \cdot \sum_{k=0}^{n-1} (w_k + w_{k+1}) \cdot (b_{k+1} - b_k)\ .$$

```
template<class RealType = double>
  class piecewise_linear_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    piecewise_linear_distribution();
    template<class InputIteratorB, class InputIteratorW>
      piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                    InputIteratorW firstW);
    template<class UnaryOperation>
      piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);
    template<class UnaryOperation>
      piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
    explicit piecewise_linear_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);
```

```
    // property functions
    vector<result_type> intervals() const;
    vector<result_type> densities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

`piecewise_linear_distribution();`

3 *Effects:* Constructs a `piecewise_linear_distribution` object with $n = 1$, $\rho_0 = \rho_1 = 1$, $b_0 = 0$, and $b_1 = 1$.

```
template<class InputIteratorB, class InputIteratorW>
 piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
                               InputIteratorW firstW);
```

4 *Mandates:* `is_invocable_r_v<double, UnaryOperation&, double>` is true.

5 ~~*Requires:*~~ *Expects:* `InputIteratorB` and `InputIteratorW` ~~shall~~ each meet the *Cpp17InputIterator* requirements (??). Moreover, the *id-expressions* `iterator_traits<InputIteratorB>::value_type` and `iterator_traits<InputIteratorW>::value_type` shall each denote a type that is convertible to `double`. If `firstB == lastB` or `++firstB == lastB`, let $n = 1$, $\rho_0 = \rho_1 = 1$, $b_0 = 0$, and $b_1 = 1$. Otherwise, $[$`firstB`, `lastB`$)$ ~~shall form~~forms a sequence b of length $n + 1$, the length of the sequence w starting from `firstW` ~~shall be~~is at least $n + 1$, and any w_k for $k \geq n + 1$ ~~shall be~~are ignored by the distribution.

6 *Effects:* Constructs a `piecewise_linear_distribution` object with parameters as specified above.

```
template<class UnaryOperation>
 piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);
```

7 *Mandates:* `is_invocable_r_v<double, UnaryOperation&, double>` is true.

8 *Requires:* Each instance of type `UnaryOperation` shall be a function object (??) whose return type shall be convertible to `double`. Moreover, `double` shall be convertible to the type of `UnaryOperation`'s sole parameter.

9 *Effects:* Constructs a `piecewise_linear_distribution` object with parameters taken or calculated from the following values: If `bl.size()` < 2, let $n = 1$, $\rho_0 = \rho_1 = 1$, $b_0 = 0$, and $b_1 = 1$. Otherwise, let $[$`bl.begin(),bl.end()`$)$ form a sequence $b_0, \dots, b_n$, and let $w_k = \texttt{fw}(b_k)$ for $k = 0, \dots, n$.

10 *Complexity:* The number of invocations of `fw` ~~shall~~does not exceed $n + 1$.

```
template<class UnaryOperation>
 piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);
```

11 *Mandates:* `is_invocable_r_v<double, UnaryOperation&, double>` is true.

12 ~~*Requires:*~~ *Expects:* ~~Each instance of type `UnaryOperation` shall be a function object (??) whose return type shall be convertible to `double`.~~ Moreover, `double` shall be convertible to the type of `UnaryOperation`'s sole parameter. If `nw` $= 0$, let $n = 1$, otherwise let $n =$ `nw`. The relation $0 < \delta = (\texttt{xmax} - \texttt{xmin})/n$ ~~shall hold~~holds.

13 *Effects:* Constructs a `piecewise_linear_distribution` object with parameters taken or calculated from the following values: Let $b_k = \texttt{xmin} + k \cdot \delta$ for $k = 0, \dots, n$, and $w_k = \texttt{fw}(b_k)$ for $k = 0, \dots, n$.

14 *Complexity:* The number of invocations of `fw` ~~shall~~does not exceed $n + 1$.

`vector<result_type> intervals() const;`

15 *Returns:* A `vector<result_type>` whose `size` member returns $n + 1$ and whose `operator[]` member returns b_k when invoked with argument k for $k = 0, \dots, n$.

`vector<result_type> densities() const;`

16 *Returns:* A `vector<result_type>` whose `size` member returns n and whose `operator[]` member returns ρ_k when invoked with argument k for $k = 0, \dots, n$.

26.6.9 Low-quality random number generation [c.math.rand]

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

```
int rand();
void srand(unsigned int seed);
```

2 *Effects:* The `rand` and `srand` functions have the semantics specified in the C standard library.

3 *Remarks:* The implementation may specify that particular library functions may call `rand`. It is implementation-defined whether the `rand` function may introduce data races (??). [*Note*: The other random number generation facilities in this document (26.6) are often preferable to `rand`, because `rand`'s underlying algorithm is unspecified. Use of `rand` therefore continues to be non-portable, with unpredictable and oft-questionable quality and performance. — *end note*]

SEE ALSO: ISO C 7.22.2

26.7 Numeric arrays [numarray]

26.7.1 Header <valarray> synopsis [valarray.syn]

```
#include <initializer_list>

namespace std {
  template<class T> class valarray;         // An array of type T
  class slice;                              // a BLAS-like slice out of an array
  template<class T> class slice_array;
  class gslice;                             // a generalized slice out of an array
  template<class T> class gslice_array;
  template<class T> class mask_array;       // a masked array
  template<class T> class indirect_array;   // an indirected array

  template<class T> void swap(valarray<T>&, valarray<T>&) noexcept;

  template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator* (const valarray<T>&,
                                           const typename valarray<T>::value_type&);
  template<class T> valarray<T> operator* (const typename valarray<T>::value_type&,
                                           const valarray<T>&);

  template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator/ (const valarray<T>&,
                                           const typename valarray<T>::value_type&);
  template<class T> valarray<T> operator/ (const typename valarray<T>::value_type&,
                                           const valarray<T>&);

  template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator% (const valarray<T>&,
                                           const typename valarray<T>::value_type&);
  template<class T> valarray<T> operator% (const typename valarray<T>::value_type&,
                                           const valarray<T>&);

  template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator+ (const valarray<T>&,
                                           const typename valarray<T>::value_type&);
  template<class T> valarray<T> operator+ (const typename valarray<T>::value_type&,
                                           const valarray<T>&);

  template<class T> valarray<T> operator- (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator- (const valarray<T>&,
                                           const typename valarray<T>::value_type&);
  template<class T> valarray<T> operator- (const typename valarray<T>::value_type&,
                                           const valarray<T>&);

  template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator^ (const valarray<T>&,
                                           const typename valarray<T>::value_type&);
```

```
template<class T> valarray<T> operator^ (const typename valarray<T>::value_type&,
                                         const valarray<T>&);

template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&,
                                         const typename valarray<T>::value_type&);
template<class T> valarray<T> operator& (const typename valarray<T>::value_type&,
                                         const valarray<T>&);

template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&,
                                         const typename valarray<T>::value_type&);
template<class T> valarray<T> operator| (const typename valarray<T>::value_type&,
                                         const valarray<T>&);

template<class T> valarray<T> operator<<(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&,
                                         const typename valarray<T>::value_type&);
template<class T> valarray<T> operator<<(const typename valarray<T>::value_type&,
                                         const valarray<T>&);

template<class T> valarray<T> operator>>(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&,
                                         const typename valarray<T>::value_type&);
template<class T> valarray<T> operator>>(const typename valarray<T>::value_type&,
                                         const valarray<T>&);

template<class T> valarray<bool> operator&&(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator&&(const typename valarray<T>::value_type&,
                                            const valarray<T>&);

template<class T> valarray<bool> operator||(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator||(const typename valarray<T>::value_type&,
                                            const valarray<T>&);

template<class T> valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator==(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator==(const typename valarray<T>::value_type&,
                                            const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator!=(const typename valarray<T>::value_type&,
                                            const valarray<T>&);

template<class T> valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator< (const typename valarray<T>::value_type&,
                                            const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator> (const typename valarray<T>::value_type&,
                                            const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
```

```
    template<class T> valarray<bool> operator<=(const typename valarray<T>::value_type&,
                                                const valarray<T>&);
    template<class T> valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
    template<class T> valarray<bool> operator>=(const valarray<T>&,
                                                const typename valarray<T>::value_type&);
    template<class T> valarray<bool> operator>=(const typename valarray<T>::value_type&,
                                                const valarray<T>&);

    template<class T> valarray<T> abs  (const valarray<T>&);
    template<class T> valarray<T> acos (const valarray<T>&);
    template<class T> valarray<T> asin (const valarray<T>&);
    template<class T> valarray<T> atan (const valarray<T>&);

    template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
    template<class T> valarray<T> atan2(const valarray<T>&,
                                        const typename valarray<T>::value_type&);
    template<class T> valarray<T> atan2(const typename valarray<T>::value_type&,
                                        const valarray<T>&);

    template<class T> valarray<T> cos  (const valarray<T>&);
    template<class T> valarray<T> cosh (const valarray<T>&);
    template<class T> valarray<T> exp  (const valarray<T>&);
    template<class T> valarray<T> log  (const valarray<T>&);
    template<class T> valarray<T> log10(const valarray<T>&);

    template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);
    template<class T> valarray<T> pow(const valarray<T>&, const typename valarray<T>::value_type&);
    template<class T> valarray<T> pow(const typename valarray<T>::value_type&, const valarray<T>&);

    template<class T> valarray<T> sin  (const valarray<T>&);
    template<class T> valarray<T> sinh (const valarray<T>&);
    template<class T> valarray<T> sqrt (const valarray<T>&);
    template<class T> valarray<T> tan  (const valarray<T>&);
    template<class T> valarray<T> tanh (const valarray<T>&);

    template<class T> unspecified1 begin(valarray<T>& v);
    template<class T> unspecified2 begin(const valarray<T>& v);
    template<class T> unspecified1 end(valarray<T>& v);
    template<class T> unspecified2 end(const valarray<T>& v);
  }
```

1 The header `<valarray>` defines five class templates (`valarray`, `slice_array`, `gslice_array`, `mask_array`, and `indirect_array`), two classes (`slice` and `gslice`), and a series of related function templates for representing and manipulating arrays of values.

2 The `valarray` array classes are defined to be free of certain forms of aliasing, thus allowing operations on these classes to be optimized.

3 Any function returning a `valarray<T>` is permitted to return an object of another type, provided all the const member functions of `valarray<T>` are also applicable to this type. This return type shall not add more than two levels of template nesting over the most deeply nested argument type.[249]

4 Implementations introducing such replacement types shall provide additional functions and operators as follows:

(4.1) — for every function taking a `const valarray<T>&` other than `begin` and `end` (26.7.10), identical functions taking the replacement types shall be added;

(4.2) — for every function taking two `const valarray<T>&` arguments, identical functions taking every combination of `const valarray<T>&` and replacement types shall be added.

5 In particular, an implementation shall allow a `valarray<T>` to be constructed from such replacement types and shall allow assignments and compound assignments of such types to `valarray<T>`, `slice_array<T>`, `gslice_array<T>`, `mask_array<T>` and `indirect_array<T>` objects.

249) **??** recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for valarray expressions.

6 These library functions are permitted to throw a `bad_alloc` (??) exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

26.7.2 Class template `valarray` [template.valarray]

26.7.2.1 Overview [template.valarray.overview]

```
namespace std {
  template<class T> class valarray {
  public:
    using value_type = T;

    // 26.7.2.2, construct/destroy
    valarray();
    explicit valarray(size_t);
    valarray(const T&, size_t);
    valarray(const T*, size_t);
    valarray(const valarray&);
    valarray(valarray&&) noexcept;
    valarray(const slice_array<T>&);
    valarray(const gslice_array<T>&);
    valarray(const mask_array<T>&);
    valarray(const indirect_array<T>&);
    valarray(initializer_list<T>);
    ~valarray();

    // 26.7.2.3, assignment
    valarray& operator=(const valarray&);
    valarray& operator=(valarray&&) noexcept;
    valarray& operator=(initializer_list<T>);
    valarray& operator=(const T&);
    valarray& operator=(const slice_array<T>&);
    valarray& operator=(const gslice_array<T>&);
    valarray& operator=(const mask_array<T>&);
    valarray& operator=(const indirect_array<T>&);

    // 26.7.2.4, element access
    const T&          operator[](size_t) const;
    T&                operator[](size_t);

    // 26.7.2.5, subset operations
    valarray          operator[](slice) const;
    slice_array<T>    operator[](slice);
    valarray          operator[](const gslice&) const;
    gslice_array<T>   operator[](const gslice&);
    valarray          operator[](const valarray<bool>&) const;
    mask_array<T>     operator[](const valarray<bool>&);
    valarray          operator[](const valarray<size_t>&) const;
    indirect_array<T> operator[](const valarray<size_t>&);

    // 26.7.2.6, unary operators
    valarray operator+() const;
    valarray operator-() const;
    valarray operator~() const;
    valarray<bool> operator!() const;

    // 26.7.2.7, compound assignment
    valarray& operator*= (const T&);
    valarray& operator/= (const T&);
    valarray& operator%= (const T&);
    valarray& operator+= (const T&);
    valarray& operator-= (const T&);
    valarray& operator^= (const T&);
    valarray& operator&= (const T&);
    valarray& operator|= (const T&);
```

```
    valarray& operator<<=(const T&);
    valarray& operator>>=(const T&);

    valarray& operator*= (const valarray&);
    valarray& operator/= (const valarray&);
    valarray& operator%= (const valarray&);
    valarray& operator+= (const valarray&);
    valarray& operator-= (const valarray&);
    valarray& operator^= (const valarray&);
    valarray& operator|= (const valarray&);
    valarray& operator&= (const valarray&);
    valarray& operator<<=(const valarray&);
    valarray& operator>>=(const valarray&);

    // 26.7.2.8, member functions
    void swap(valarray&) noexcept;

    size_t size() const;

    T sum() const;
    T min() const;
    T max() const;

    valarray shift (int) const;
    valarray cshift(int) const;
    valarray apply(T func(T)) const;
    valarray apply(T func(const T&)) const;
    void resize(size_t sz, T c = T());
  };

  template<class T, size_t cnt> valarray(const T(&)[cnt], size_t) -> valarray<T>;
}
```

1 The class template `valarray<T>` is a one-dimensional smart array, with elements numbered sequentially from zero. It is a representation of the mathematical concept of an ordered set of values. For convenience, an object of type `valarray<T>` is referred to as an "array" throughout the remainder of 26.7. The illusion of higher dimensionality may be produced by the familiar idiom of computed indices, together with the powerful subsetting capabilities provided by the generalized subscript operators.[250]

26.7.2.2 Constructors [valarray.cons]

`valarray();`

1 *Effects:* Constructs a `valarray` that has zero length.[251]

`explicit valarray(size_t n);`

2 *Effects:* Constructs a `valarray` that has length `n`. Each element of the array is value-initialized (**??**).

`valarray(const T& v, size_t n);`

3 *Effects:* Constructs a `valarray` that has length `n`. Each element of the array is initialized with `v`.

`valarray(const T* p, size_t n);`

4 *Requires:* `p` points to an array (**??**) of at least `n` elements.

5 *Expects:* `[p, p + n)` is a valid range.

Effects: Constructs a `valarray` that has length `n`. The values of the elements of the array are initialized with the first `n` values pointed to by the first argument.[252]

250) The intent is to specify an array template that has the minimum functionality necessary to address aliasing ambiguities and the proliferation of temporary objects. Thus, the `valarray` template is neither a matrix class nor a field class. However, it is a very useful building block for designing such classes.
251) This default constructor is essential, since arrays of `valarray` may be useful. After initialization, the length of an empty array can be increased with the `resize` member function.
252) This constructor is the preferred method for converting a C array to a `valarray` object.

```
valarray(const valarray& v);
```

6 *Effects:* Constructs a `valarray` that has the same length as `v`. The elements are initialized with the values of the corresponding elements of `v`.[253]

```
valarray(valarray&& v) noexcept;
```

7 *Effects:* Constructs a `valarray` that has the same length as `v`. The elements are initialized with the values of the corresponding elements of `v`.

8 *Complexity:* Constant.

```
valarray(initializer_list<T> il);
```

9 *Effects:* Equivalent to `valarray(il.begin(), il.size())`.

```
valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);
```

10 These conversion constructors convert one of the four reference templates to a `valarray`.

```
~valarray();
```

11 *Effects:* The destructor is applied to every element of `*this`; an implementation may return all allocated memory.

26.7.2.3 Assignment [valarray.assign]

```
valarray& operator=(const valarray& v);
```

1 *Effects:* Each element of the `*this` array is assigned the value of the corresponding element of `v`. If the length of `v` is not equal to the length of `*this`, resizes `*this` to make the two arrays the same length, as if by calling `resize(v.size())`, before performing the assignment.

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

3 *Returns:* `*this`.

```
valarray& operator=(valarray&& v) noexcept;
```

4 *Effects:* `*this` obtains the value of `v`. The value of `v` after the assignment is not specified.

5 *Returns:* `*this`.

6 *Complexity:* Linear.

```
valarray& operator=(initializer_list<T> il);
```

7 *Effects:* Equivalent to: `return *this = valarray(il);`

```
valarray& operator=(const T& v);
```

8 *Effects:* Assigns `v` to each element of `*this`.

9 *Returns:* `*this`.

```
valarray& operator=(const slice_array<T>&);
valarray& operator=(const gslice_array<T>&);
valarray& operator=(const mask_array<T>&);
valarray& operator=(const indirect_array<T>&);
```

10 ~~*Requires:*~~ *Expects:* The length of the array to which the argument refers equals `size()`. The value of an element in the left-hand side of a `valarray` assignment operator does not depend on the value of another element in that left-hand side.

11 These operators allow the results of a generalized subscripting operation to be assigned directly to a `valarray`.

253) This copy constructor creates a distinct array rather than an alias. Implementations in which arrays share storage are permitted, but they would need to implement a copy-on-reference mechanism to ensure that arrays are conceptually distinct.

26.7.2.4 Element access [valarray.access]

```
const T&  operator[](size_t n) const;
T& operator[](size_t n);
```

1 ~~*Requires:*~~ *Expects:* `n < size()` is `true`.

2 *Returns:* A reference to the corresponding element of the array. [*Note*: The expression `(a[i] = q, a[i]) == q` evaluates to `true` for any non-constant `valarray<T> a`, any `T q`, and for any `size_t i` such that the value of `i` is less than the length of `a`. — *end note*]

3 *Remarks:* The expression `addressof(a[i+j]) == addressof(a[i]) + j` evaluates to `true` for all `size_t i` and `size_t j` such that `i+j < a.size()`.

4 The expression `addressof(a[i]) != addressof(b[j])` evaluates to `true` for any two arrays `a` and `b` and for any `size_t i` and `size_t j` such that `i < a.size()` and `j < b.size()`. [*Note*: This property indicates an absence of aliasing and may be used to advantage by optimizing compilers. Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from `operator new`, and other techniques to generate efficient `valarray`s. — *end note*]

5 The reference returned by the subscript operator for an array shall be valid until the member function `resize(size_t, T)` (26.7.2.8) is called for that array or until the lifetime of that array ends, whichever happens first.

26.7.2.5 Subset operations [valarray.sub]

1 The member `operator[]` is overloaded to provide several ways to select sequences of elements from among those controlled by `*this`. Each of these operations returns a subset of the array. The const-qualified versions return this subset as a new `valarray` object. The non-const versions return a class template object which has reference semantics to the original array, working in conjunction with various overloads of `operator=` and other assigning operators to allow selective replacement (slicing) of the controlled sequence. In each case the selected element(s) shall exist.

```
valarray operator[](slice slicearr) const;
```

2 *Returns:* A `valarray` containing those elements of the controlled sequence designated by `slicearr`. [*Example*:

```
const valarray<char> v0("abcdefghijklmnop", 16);
// v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
```

— *end example*]

```
slice_array<T> operator[](slice slicearr);
```

3 *Returns:* An object that holds references to elements of the controlled sequence selected by `slicearr`. [*Example*:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
v0[slice(2, 5, 3)] = v1;
// v0 == valarray<char>("abAdeBghCjkDmnEp", 16);
```

— *end example*]

```
valarray operator[](const gslice& gslicearr) const;
```

4 *Returns:* A `valarray` containing those elements of the controlled sequence designated by `gslicearr`. [*Example*:

```
const valarray<char> v0("abcdefghijklmnop", 16);
const size_t lv[] = { 2, 3 };
const size_t dv[] = { 7, 2 };
const valarray<size_t> len(lv, 2), str(dv, 2);
// v0[gslice(3, len, str)] returns
// valarray<char>("dfhkmo", 6)
```

— *end example*]

```
gslice_array<T> operator[](const gslice& gslicearr);
```

5 *Returns:* An object that holds references to elements of the controlled sequence selected by `gslicearr`. [*Example*:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDEF", 6);
const size_t lv[] = { 2, 3 };
const size_t dv[] = { 7, 2 };
const valarray<size_t> len(lv, 2), str(dv, 2);
v0[gslice(3, len, str)] = v1;
// v0 == valarray<char>("abcAeBgCijDlEnFp", 16)
```

— *end example*]

```
valarray operator[](const valarray<bool>& boolarr) const;
```

6 *Returns:* A `valarray` containing those elements of the controlled sequence designated by `boolarr`. [*Example*:

```
const valarray<char> v0("abcdefghijklmnop", 16);
const bool vb[] = { false, false, true, true, false, true };
// v0[valarray<bool>(vb, 6)] returns
// valarray<char>("cdf", 3)
```

— *end example*]

```
mask_array<T> operator[](const valarray<bool>& boolarr);
```

7 *Returns:* An object that holds references to elements of the controlled sequence selected by `boolarr`. [*Example*:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABC", 3);
const bool vb[] = { false, false, true, true, false, true };
v0[valarray<bool>(vb, 6)] = v1;
// v0 == valarray<char>("abABeCghijklmnop", 16)
```

— *end example*]

```
valarray operator[](const valarray<size_t>& indarr) const;
```

8 *Returns:* A `valarray` containing those elements of the controlled sequence designated by `indarr`. [*Example*:

```
const valarray<char> v0("abcdefghijklmnop", 16);
const size_t vi[] = { 7, 5, 2, 3, 8 };
// v0[valarray<size_t>(vi, 5)] returns
// valarray<char>("hfcdi", 5)
```

— *end example*]

```
indirect_array<T> operator[](const valarray<size_t>& indarr);
```

9 *Returns:* An object that holds references to elements of the controlled sequence selected by `indarr`. [*Example*:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
const size_t vi[] = { 7, 5, 2, 3, 8 };
v0[valarray<size_t>(vi, 5)] = v1;
// v0 == valarray<char>("abCDeBgAEjklmnop", 16)
```

— *end example*]

26.7.2.6 Unary operators [valarray.unary]

```
valarray operator+() const;
valarray operator-() const;
valarray operator~() const;
valarray<bool> operator!() const;
```

1 *Mandates:* The indicated operator can be applied to operands of type `T` and returns a value of type `T` (`bool` for `operator!`) or which may be unambiguously implicitly converted to type `T` (`bool` for `operator!`).

2 *Requires:* Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` (`bool` for `operator!`) or which may be unambiguously implicitly converted to type `T` (`bool` for `operator!`).

3 *Returns:* A `valarray` whose length is `size()`. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

26.7.2.7 Compound assignment [valarray.cassign]

```
valarray& operator*= (const valarray& v);
valarray& operator/= (const valarray& v);
valarray& operator%= (const valarray& v);
valarray& operator+= (const valarray& v);
valarray& operator-= (const valarray& v);
valarray& operator^= (const valarray& v);
valarray& operator&= (const valarray& v);
valarray& operator|= (const valarray& v);
valarray& operator<<=(const valarray& v);
valarray& operator>>=(const valarray& v);
```

1 *Mandates:* The indicated operator can be applied to two operands of type `T`.

2 ~~*Requires:*~~ *Expects:* `size() == v.size()` is `true`. ~~Each of these operators may only be instantiated for a type `T` if the indicated operator can be applied to two operands of type `T`.~~ The value of an element in the left-hand side of a valarray compound assignment operator does not depend on the value of another element in that left hand side.

3 *Effects:* Each of these operators performs the indicated operation on each of the elements of `*this` and the corresponding element of `v`.

4 *Returns:* `*this`.

5 *Remarks:* The appearance of an array on the left-hand side of a compound assignment does not invalidate references or pointers.

```
valarray& operator*= (const T& v);
valarray& operator/= (const T& v);
valarray& operator%= (const T& v);
valarray& operator+= (const T& v);
valarray& operator-= (const T& v);
valarray& operator^= (const T& v);
valarray& operator&= (const T& v);
valarray& operator|= (const T& v);
valarray& operator<<=(const T& v);
valarray& operator>>=(const T& v);
```

6 *Mandates:* The indicated operator can be applied to two operands of type `T`.

7 *Requires:* Each of these operators may only be instantiated for a type `T` if the indicated operator can be applied to two operands of type `T`.

8 *Effects:* Each of these operators applies the indicated operation to each element of `*this` and `v`.

9 *Returns:* `*this`

10 *Remarks:* The appearance of an array on the left-hand side of a compound assignment does not invalidate references or pointers to the elements of the array.

26.7.2.8 Member functions [valarray.members]

```
void swap(valarray& v) noexcept;
```

1 *Effects:* `*this` obtains the value of `v`. `v` obtains the value of `*this`.

2 *Complexity:* Constant.

```
size_t size() const;
```

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

4 *Complexity:* Constant time.

```
T sum() const;
```

5 *Mandates:* `operator+=` can be applied to operands of type `T`.

6 ~~*Requires:*~~ *Expects:* `size() > 0` is `true`. ~~This function may only be instantiated for a type `T` to which `operator+=` can be applied.~~

7 *Returns:* The sum of all the elements of the array. If the array has length 1, returns the value of element 0. Otherwise, the returned value is calculated by applying `operator+=` to a copy of an element of the array and all other elements of the array in an unspecified order.

```
T min() const;
```

8 ~~*Requires:*~~ *Expects:* `size() > 0` is `true`.

9 *Returns:* The minimum value contained in `*this`. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator<`.

```
T max() const;
```

10 ~~*Requires:*~~ *Expects:* `size() > 0` is `true`.

11 *Returns:* The maximum value contained in `*this`. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator<`.

```
valarray shift(int n) const;
```

12 *Returns:* A `valarray` of length `size()`, each of whose elements I is `(*this)[`I` + n]` if I `+ n` is non-negative and less than `size()`, otherwise `T()`. [*Note*: If element zero is taken as the leftmost element, a positive value of `n` shifts the elements left `n` places, with zero fill. — *end note*]

13 [*Example*: If the argument has the value -2, the first two elements of the result will be value-initialized (**??**); the third element of the result will be assigned the value of the first element of the argument; etc. — *end example*]

```
valarray cshift(int n) const;
```

14 *Returns:* A `valarray` of length `size()` that is a circular shift of `*this`. If element zero is taken as the leftmost element, a non-negative value of n shifts the elements circularly left n places and a negative value of n shifts the elements circularly right $-n$ places.

```
valarray apply(T func(T)) const;
valarray apply(T func(const T&)) const;
```

15 *Returns:* A `valarray` whose length is `size()`. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of `*this`.

```
void resize(size_t sz, T c = T());
```

16 *Effects:* Changes the length of the `*this` array to `sz` and then assigns to each element the value of the second argument. Resizing invalidates all pointers and references to elements in the array.

26.7.3 valarray non-member operations [valarray.nonmembers]

26.7.3.1 Binary operators [valarray.binary]

```
template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> operator>>(const valarray<T>&, const valarray<T>&);
```

1 *Mandates:* The indicated operator can be applied to operands of type `T` and returns a value of type `T` or which can be unambiguously implicitly converted to `T`.

2 ~~*Requires:*~~ *Expects:* ~~Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` or which can be unambiguously implicitly converted to type `T`.~~ The argument arrays have the same length.

3 *Returns:* A `valarray` whose length is equal to the lengths of the argument arrays. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

```
template<class T> valarray<T> operator* (const valarray<T>&,
                                          const typename valarray<T>::value_type&);
template<class T> valarray<T> operator* (const typename valarray<T>::value_type&,
                                          const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&,
                                          const typename valarray<T>::value_type&);
template<class T> valarray<T> operator/ (const typename valarray<T>::value_type&,
                                          const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&,
                                          const typename valarray<T>::value_type&);
template<class T> valarray<T> operator% (const typename valarray<T>::value_type&,
                                          const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&,
                                          const typename valarray<T>::value_type&);
template<class T> valarray<T> operator+ (const typename valarray<T>::value_type&,
                                          const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&,
                                          const typename valarray<T>::value_type&);
template<class T> valarray<T> operator- (const typename valarray<T>::value_type&,
                                          const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&,
                                          const typename valarray<T>::value_type&);
template<class T> valarray<T> operator^ (const typename valarray<T>::value_type&,
                                          const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&,
                                          const typename valarray<T>::value_type&);
template<class T> valarray<T> operator& (const typename valarray<T>::value_type&,
                                          const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&,
                                          const typename valarray<T>::value_type&);
template<class T> valarray<T> operator| (const typename valarray<T>::value_type&,
                                          const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&,
                                          const typename valarray<T>::value_type&);
template<class T> valarray<T> operator<<(const typename valarray<T>::value_type&,
                                          const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&,
                                          const typename valarray<T>::value_type&);
template<class T> valarray<T> operator>>(const typename valarray<T>::value_type&,
                                          const valarray<T>&);
```

4 *Mandates:* The indicated operator can be applied to operands of type `T` and returns a value of type `T` or which can be unambiguously implicitly converted to `T`.

5 *Requires:* Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` or which can be unambiguously implicitly converted to type `T`.

6 *Returns:* A `valarray` whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the non-array argument.

26.7.3.2 Logical operators [valarray.comparison]

```
template<class T> valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&, const valarray<T>&);
```

1 *Mandates:* The indicated operator can be applied to operands of type `T` and returns a value of type `bool` or which can be unambiguously implicitly converted to `bool`.

2 ~~*Requires:*~~ *Expects:* ~~Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `bool` or which can be unambiguously implicitly converted to type `bool`.~~ The two array arguments have the same length.

3 *Returns:* A `valarray<bool>` whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

```
template<class T> valarray<bool> operator==(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator==(const typename valarray<T>::value_type&,
                                            const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator!=(const typename valarray<T>::value_type&,
                                            const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator< (const typename valarray<T>::value_type&,
                                            const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator> (const typename valarray<T>::value_type&,
                                            const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator<=(const typename valarray<T>::value_type&,
                                            const valarray<T>&);
template<class T> valarray<bool> operator>=(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator>=(const typename valarray<T>::value_type&,
                                            const valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator&&(const typename valarray<T>::value_type&,
                                            const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&,
                                            const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator||(const typename valarray<T>::value_type&,
                                            const valarray<T>&);
```

4 *Mandates:* The indicated operator can be applied to operands of type `T` and returns a value of type `bool` or which can be unambiguously implicitly converted to `bool`.

5 *Requires:* Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `bool` or which can be unambiguously implicitly converted to type `bool`.

6 *Returns:* A `valarray<bool>` whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array and the non-array argument.

26.7.3.3 Transcendentals [valarray.transcend]

```
template<class T> valarray<T> abs  (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> atan2(const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<T> cos  (const valarray<T>&);
template<class T> valarray<T> cosh (const valarray<T>&);
template<class T> valarray<T> exp  (const valarray<T>&);
template<class T> valarray<T> log  (const valarray<T>&);
template<class T> valarray<T> log10(const valarray<T>&);
template<class T> valarray<T> pow  (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> pow  (const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> pow  (const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<T> sin  (const valarray<T>&);
template<class T> valarray<T> sinh (const valarray<T>&);
template<class T> valarray<T> sqrt (const valarray<T>&);
template<class T> valarray<T> tan  (const valarray<T>&);
template<class T> valarray<T> tanh (const valarray<T>&);
```

1 *Mandates:* A unique function with the indicated name can be applied (unqualified) to operands of type `T`. This function returns a value of type `T` or which can be unambiguously implicitly converted to type `T`.

2 *Requires:* Each of these functions may only be instantiated for a type `T` to which a unique function with the indicated name can be applied (unqualified). This function shall return a value of type `T` or which can be unambiguously implicitly converted to type `T`.

26.7.3.4 Specialized algorithms [valarray.special]

```
template<class T> void swap(valarray<T>& x, valarray<T>& y) noexcept;
```

1 *Effects:* Equivalent to `x.swap(y)`.

26.7.4 Class slice [class.slice]

26.7.4.1 Overview [class.slice.overview]

```
namespace std {
  class slice {
  public:
    slice();
    slice(size_t, size_t, size_t);

    size_t start() const;
    size_t size() const;
    size_t stride() const;

    friend bool operator==(const slice& x, const slice& y);
  };
}
```

1 The `slice` class represents a BLAS-like slice from an array. Such a slice is specified by a starting index, a length, and a stride.[254]

26.7.4.2 Constructors [cons.slice]

```
slice();
slice(size_t start, size_t length, size_t stride);
```

254) BLAS stands for *Basic Linear Algebra Subprograms.* C++ programs may instantiate this class. See, for example, Dongarra, Du Croz, Duff, and Hammerling: *A set of Level 3 Basic Linear Algebra Subprograms*; Technical Report MCS-P1-0888, Argonne National Laboratory (USA), Mathematics and Computer Science Division, August, 1988.

```
slice(const slice&);
```

1 The default constructor is equivalent to `slice(0, 0, 0)`. A default constructor is provided only to permit the declaration of arrays of slices. The constructor with arguments for a slice takes a start, length, and stride parameter.

2 [*Example*: `slice(3, 8, 2)` constructs a slice which selects elements $3, 5, 7, \ldots, 17$ from an array. *— end example*]

26.7.4.3 Access functions [slice.access]

```
size_t start() const;
size_t size() const;
size_t stride() const;
```

1 *Returns:* The start, length, or stride specified by a `slice` object.

2 *Complexity:* Constant time.

26.7.4.4 Operators [slice.ops]

```
friend bool operator==(const slice& x, const slice& y);
```

1 *Effects:* Equivalent to:

```
return x.start() == y.start() && x.size() == y.size() && x.stride() == y.stride();
```

26.7.5 Class template `slice_array` [template.slice.array]

26.7.5.1 Overview [template.slice.array.overview]

```
namespace std {
  template<class T> class slice_array {
  public:
    using value_type = T;

    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    slice_array(const slice_array&);
    ~slice_array();
    const slice_array& operator=(const slice_array&) const;
    void operator=(const T&) const;

    slice_array() = delete;       // as implied by declaring copy constructor above
  };
}
```

1 This template is a helper template used by the `slice` subscript operator

```
slice_array<T> valarray<T>::operator[](slice);
```

2 It has reference semantics to a subset of an array specified by a `slice` object. [*Example*: The expression `a[slice(1, 5, 3)] = b;` has the effect of assigning the elements of `b` to a slice of the elements in `a`. For the slice shown, the elements selected from `a` are $1, 4, \ldots, 13$. *— end example*]

26.7.5.2 Assignment [slice.arr.assign]

```
void operator=(const valarray<T>&) const;
const slice_array& operator=(const slice_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `slice_array` object refers.

26.7.5.3 Compound assignment [slice.arr.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These compound assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `slice_array` object refers.

26.7.5.4 Fill function [slice.arr.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `slice_array` object refers.

26.7.6 The `gslice` class [class.gslice]

26.7.6.1 Overview [class.gslice.overview]

```
namespace std {
  class gslice {
  public:
    gslice();
    gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

    size_t           start() const;
    valarray<size_t> size() const;
    valarray<size_t> stride() const;
  };
}
```

1 This class represents a generalized slice out of an array. A `gslice` is defined by a starting offset (s), a set of lengths (l_j), and a set of strides (d_j). The number of lengths shall equal the number of strides.

2 A `gslice` represents a mapping from a set of indices (i_j), equal in number to the number of strides, to a single index k. It is useful for building multidimensional array classes using the `valarray` template, which is one-dimensional. The set of one-dimensional index values specified by a `gslice` are

$$k = s + \sum_j i_j d_j$$

where the multidimensional indices i_j range in value from 0 to $l_{ij} - 1$.

3 [*Example*: The `gslice` specification

```
start  = 3
length = {2, 4, 3}
stride = {19, 4, 1}
```

yields the sequence of one-dimensional indices

$$k = 3 + (0, 1) \times 19 + (0, 1, 2, 3) \times 4 + (0, 1, 2) \times 1$$

which are ordered as shown in the following table:

$$\begin{array}{lllll} (i_0, & i_1, & i_2, & k) & = \\ & (0, & 0, & 0, & 3), \\ & (0, & 0, & 1, & 4), \\ & (0, & 0, & 2, & 5), \\ & (0, & 1, & 0, & 7), \end{array}$$

(0, 1, 1, 8),
(0, 1, 2, 9),
(0, 2, 0, 11),
(0, 2, 1, 12),
(0, 2, 2, 13),
(0, 3, 0, 15),
(0, 3, 1, 16),
(0, 3, 2, 17),
(1, 0, 0, 22),
(1, 0, 1, 23),
. . .
(1, 3, 2, 36)

That is, the highest-ordered index turns fastest. — *end example*]

4 It is possible to have degenerate generalized slices in which an address is repeated.

5 [*Example*: If the stride parameters in the previous example are changed to {1, 1, 1}, the first few elements of the resulting sequence of indices will be

(0, 0, 0, 3),
(0, 0, 1, 4),
(0, 0, 2, 5),
(0, 1, 0, 4),
(0, 1, 1, 5),
(0, 1, 2, 6),
. . .

— *end example*]

6 If a degenerate slice is used as the argument to the non-`const` version of `operator[](const gslice&)`, the behavior is undefined.

26.7.6.2 Constructors [gslice.cons]

```
gslice();
gslice(size_t start, const valarray<size_t>& lengths,
         const valarray<size_t>& strides);
gslice(const gslice&);
```

1 The default constructor is equivalent to `gslice(0, valarray<size_t>(), valarray<size_t>())`. The constructor with arguments builds a `gslice` based on a specification of start, lengths, and strides, as explained in the previous subclause.

26.7.6.3 Access functions [gslice.access]

```
size_t           start()  const;
valarray<size_t> size() const;
valarray<size_t> stride() const;
```

1 *Returns:* The representation of the start, lengths, or strides specified for the `gslice`.

2 *Complexity:* `start()` is constant time. `size()` and `stride()` are linear in the number of strides.

26.7.7 Class template gslice_array [template.gslice.array]

26.7.7.1 Overview [template.gslice.array.overview]

```
namespace std {
  template<class T> class gslice_array {
  public:
    using value_type = T;
```

```
    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    gslice_array(const gslice_array&);
    ~gslice_array();
    const gslice_array& operator=(const gslice_array&) const;
    void operator=(const T&) const;

    gslice_array() = delete;       // as implied by declaring copy constructor above
  };
}
```

1 This template is a helper template used by the `gslice` subscript operator

```
gslice_array<T> valarray<T>::operator[](const gslice&);
```

2 It has reference semantics to a subset of an array specified by a `gslice` object. Thus, the expression `a[gslice(1, length, stride)] = b` has the effect of assigning the elements of `b` to a generalized slice of the elements in `a`.

26.7.7.2 Assignment [gslice.array.assign]

```
void operator=(const valarray<T>&) const;
const gslice_array& operator=(const gslice_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `gslice_array` refers.

26.7.7.3 Compound assignment [gslice.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These compound assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `gslice_array` object refers.

26.7.7.4 Fill function [gslice.array.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `gslice_array` object refers.

26.7.8 Class template `mask_array` [template.mask.array]

26.7.8.1 Overview [template.mask.array.overview]

```
namespace std {
  template<class T> class mask_array {
  public:
    using value_type = T;
```

```
    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    mask_array(const mask_array&);
    ~mask_array();
    const mask_array& operator=(const mask_array&) const;
    void operator=(const T&) const;

    mask_array() = delete;        // as implied by declaring copy constructor above
  };
}
```

1 This template is a helper template used by the mask subscript operator:

`mask_array<T> valarray<T>::operator[](const valarray<bool>&).`

2 It has reference semantics to a subset of an array specified by a boolean mask. Thus, the expression `a[mask] = b;` has the effect of assigning the elements of `b` to the masked elements in `a` (those for which the corresponding element in `mask` is `true`).

26.7.8.2 Assignment [mask.array.assign]

```
void operator=(const valarray<T>&) const;
const mask_array& operator=(const mask_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

26.7.8.3 Compound assignment [mask.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These compound assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the mask object refers.

26.7.8.4 Fill function [mask.array.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `mask_array` object refers.

26.7.9 Class template `indirect_array` [template.indirect.array]

26.7.9.1 Overview [template.indirect.array.overview]

```
namespace std {
  template<class T> class indirect_array {
  public:
    using value_type = T;
```

```
    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    indirect_array(const indirect_array&);
    ~indirect_array();
    const indirect_array& operator=(const indirect_array&) const;
    void operator=(const T&) const;

    indirect_array() = delete;        // as implied by declaring copy constructor above
  };
}
```

1 This template is a helper template used by the indirect subscript operator

`indirect_array<T> valarray<T>::operator[](const valarray<size_t>&).`

2 It has reference semantics to a subset of an array specified by an `indirect_array`. Thus, the expression `a[indirect] = b;` has the effect of assigning the elements of `b` to the elements in `a` whose indices appear in `indirect`.

26.7.9.2 Assignment [indirect.array.assign]

```
void operator=(const valarray<T>&) const;
const indirect_array& operator=(const indirect_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

2 If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

3 [*Example*:

```
int addr[] = {2, 3, 1, 4, 4};
valarray<size_t> indirect(addr, 5);
valarray<double> a(0., 10), b(1., 5);
a[indirect] = b;
```

results in undefined behavior since element 4 is specified twice in the indirection. — *end example*]

26.7.9.3 Compound assignment [indirect.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These compound assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `indirect_array` object refers.

2 If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

26.7.9.4 Fill function [indirect.array.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `indirect_array` object refers.

26.7.10 valarray range access [valarray.range]

1 In the `begin` and `end` function templates that follow, *`unspecified`*1 is a type that meets the requirements of a mutable *Cpp17RandomAccessIterator* (??) and models `contiguous_iterator` (??), whose `value_type` is the template parameter `T` and whose `reference` type is `T&`. *`unspecified`*2 is a type that meets the requirements of a constant *Cpp17RandomAccessIterator* and models `contiguous_iterator`, whose `value_type` is the template parameter `T` and whose `reference` type is `const T&`.

2 The iterators returned by `begin` and `end` for an array are guaranteed to be valid until the member function `resize(size_t, T)` (26.7.2.8) is called for that array or until the lifetime of that array ends, whichever happens first.

```
template<class T> unspecified1 begin(valarray<T>& v);
template<class T> unspecified2 begin(const valarray<T>& v);
```

3 *Returns:* An iterator referencing the first value in the array.

```
template<class T> unspecified1 end(valarray<T>& v);
template<class T> unspecified2 end(const valarray<T>& v);
```

4 *Returns:* An iterator referencing one past the last value in the array.

26.8 Mathematical functions for floating-point types [c.math]

26.8.1 Header <cmath> synopsis [cmath.syn]

```
namespace std {
  using float_t = see below;
  using double_t = see below;
}

#define HUGE_VAL see below
#define HUGE_VALF see below
#define HUGE_VALL see below
#define INFINITY see below
#define NAN see below
#define FP_INFINITE see below
#define FP_NAN see below
#define FP_NORMAL see below
#define FP_SUBNORMAL see below
#define FP_ZERO see below
#define FP_FAST_FMA see below
#define FP_FAST_FMAF see below
#define FP_FAST_FMAL see below
#define FP_ILOGB0 see below
#define FP_ILOGBNAN see below
#define MATH_ERRNO see below
#define MATH_ERREXCEPT see below

#define math_errhandling see below

namespace std {
  float acos(float x);  // see ??
  double acos(double x);
  long double acos(long double x);  // see ??
  float acosf(float x);
  long double acosl(long double x);

  float asin(float x);  // see ??
  double asin(double x);
  long double asin(long double x);  // see ??
```

```
float asinf(float x);
long double asinl(long double x);

float atan(float x);  // see ??
double atan(double x);
long double atan(long double x);  // see ??
float atanf(float x);
long double atanl(long double x);

float atan2(float y, float x);  // see ??
double atan2(double y, double x);
long double atan2(long double y, long double x);  // see ??
float atan2f(float y, float x);
long double atan2l(long double y, long double x);

float cos(float x);  // see ??
double cos(double x);
long double cos(long double x);  // see ??
float cosf(float x);
long double cosl(long double x);

float sin(float x);  // see ??
double sin(double x);
long double sin(long double x);  // see ??
float sinf(float x);
long double sinl(long double x);

float tan(float x);  // see ??
double tan(double x);
long double tan(long double x);  // see ??
float tanf(float x);
long double tanl(long double x);

float acosh(float x);  // see ??
double acosh(double x);
long double acosh(long double x);  // see ??
float acoshf(float x);
long double acoshl(long double x);

float asinh(float x);  // see ??
double asinh(double x);
long double asinh(long double x);  // see ??
float asinhf(float x);
long double asinhl(long double x);

float atanh(float x);  // see ??
double atanh(double x);
long double atanh(long double x);  // see ??
float atanhf(float x);
long double atanhl(long double x);

float cosh(float x);  // see ??
double cosh(double x);
long double cosh(long double x);  // see ??
float coshf(float x);
long double coshl(long double x);

float sinh(float x);  // see ??
double sinh(double x);
long double sinh(long double x);  // see ??
float sinhf(float x);
long double sinhl(long double x);
```

```
float tanh(float x);  // see ??
double tanh(double x);
long double tanh(long double x);  // see ??
float tanhf(float x);
long double tanhl(long double x);

float exp(float x);  // see ??
double exp(double x);
long double exp(long double x);  // see ??
float expf(float x);
long double expl(long double x);

float exp2(float x);  // see ??
double exp2(double x);
long double exp2(long double x);  // see ??
float exp2f(float x);
long double exp2l(long double x);

float expm1(float x);  // see ??
double expm1(double x);
long double expm1(long double x);  // see ??
float expm1f(float x);
long double expm1l(long double x);

float frexp(float value, int* exp);  // see ??
double frexp(double value, int* exp);
long double frexp(long double value, int* exp);  // see ??
float frexpf(float value, int* exp);
long double frexpl(long double value, int* exp);

int ilogb(float x);  // see ??
int ilogb(double x);
int ilogb(long double x);  // see ??
int ilogbf(float x);
int ilogbl(long double x);

float ldexp(float x, int exp);  // see ??
double ldexp(double x, int exp);
long double ldexp(long double x, int exp);  // see ??
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);

float log(float x);  // see ??
double log(double x);
long double log(long double x);  // see ??
float logf(float x);
long double logl(long double x);

float log10(float x);  // see ??
double log10(double x);
long double log10(long double x);  // see ??
float log10f(float x);
long double log10l(long double x);

float log1p(float x);  // see ??
double log1p(double x);
long double log1p(long double x);  // see ??
float log1pf(float x);
long double log1pl(long double x);

float log2(float x);  // see ??
double log2(double x);
long double log2(long double x);  // see ??
float log2f(float x);
```

```
long double log2l(long double x);

float logb(float x);  // see ??
double logb(double x);
long double logb(long double x);  // see ??
float logbf(float x);
long double logbl(long double x);

float modf(float value, float* iptr);  // see ??
double modf(double value, double* iptr);
long double modf(long double value, long double* iptr);  // see ??
float modff(float value, float* iptr);
long double modfl(long double value, long double* iptr);

float scalbn(float x, int n);  // see ??
double scalbn(double x, int n);
long double scalbn(long double x, int n);  // see ??
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);

float scalbln(float x, long int n);  // see ??
double scalbln(double x, long int n);
long double scalbln(long double x, long int n);  // see ??
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);

float cbrt(float x);  // see ??
double cbrt(double x);
long double cbrt(long double x);  // see ??
float cbrtf(float x);
long double cbrtl(long double x);

// 26.8.2, absolute values
int abs(int j);
long int abs(long int j);
long long int abs(long long int j);
float abs(float j);
double abs(double j);
long double abs(long double j);

float fabs(float x);  // see ??
double fabs(double x);
long double fabs(long double x);  // see ??
float fabsf(float x);
long double fabsl(long double x);

float hypot(float x, float y);  // see ??
double hypot(double x, double y);
long double hypot(long double x, long double y);  // see ??
float hypotf(float x, float y);
long double hypotl(long double x, long double y);

// 26.8.3, three-dimensional hypotenuse
float hypot(float x, float y, float z);
double hypot(double x, double y, double z);
long double hypot(long double x, long double y, long double z);

float pow(float x, float y);  // see ??
double pow(double x, double y);
long double pow(long double x, long double y);  // see ??
float powf(float x, float y);
long double powl(long double x, long double y);
```

```
float sqrt(float x);  // see ??
double sqrt(double x);
long double sqrt(long double x);  // see ??
float sqrtf(float x);
long double sqrtl(long double x);

float erf(float x);  // see ??
double erf(double x);
long double erf(long double x);  // see ??
float erff(float x);
long double erfl(long double x);

float erfc(float x);  // see ??
double erfc(double x);
long double erfc(long double x);  // see ??
float erfcf(float x);
long double erfcl(long double x);

float lgamma(float x);  // see ??
double lgamma(double x);
long double lgamma(long double x);  // see ??
float lgammaf(float x);
long double lgammal(long double x);

float tgamma(float x);  // see ??
double tgamma(double x);
long double tgamma(long double x);  // see ??
float tgammaf(float x);
long double tgammal(long double x);

float ceil(float x);  // see ??
double ceil(double x);
long double ceil(long double x);  // see ??
float ceilf(float x);
long double ceill(long double x);

float floor(float x);  // see ??
double floor(double x);
long double floor(long double x);  // see ??
float floorf(float x);
long double floorl(long double x);

float nearbyint(float x);  // see ??
double nearbyint(double x);
long double nearbyint(long double x);  // see ??
float nearbyintf(float x);
long double nearbyintl(long double x);

float rint(float x);  // see ??
double rint(double x);
long double rint(long double x);  // see ??
float rintf(float x);
long double rintl(long double x);

long int lrint(float x);  // see ??
long int lrint(double x);
long int lrint(long double x);  // see ??
long int lrintf(float x);
long int lrintl(long double x);

long long int llrint(float x);  // see ??
long long int llrint(double x);
long long int llrint(long double x);  // see ??
long long int llrintf(float x);
```

```
long long int llrintl(long double x);

float round(float x);  // see ??
double round(double x);
long double round(long double x);  // see ??
float roundf(float x);
long double roundl(long double x);

long int lround(float x);  // see ??
long int lround(double x);
long int lround(long double x);  // see ??
long int lroundf(float x);
long int lroundl(long double x);

long long int llround(float x);  // see ??
long long int llround(double x);
long long int llround(long double x);  // see ??
long long int llroundf(float x);
long long int llroundl(long double x);

float trunc(float x);  // see ??
double trunc(double x);
long double trunc(long double x);  // see ??
float truncf(float x);
long double truncl(long double x);

float fmod(float x, float y);  // see ??
double fmod(double x, double y);
long double fmod(long double x, long double y);  // see ??
float fmodf(float x, float y);
long double fmodl(long double x, long double y);

float remainder(float x, float y);  // see ??
double remainder(double x, double y);
long double remainder(long double x, long double y);  // see ??
float remainderf(float x, float y);
long double remainderl(long double x, long double y);

float remquo(float x, float y, int* quo);  // see ??
double remquo(double x, double y, int* quo);
long double remquo(long double x, long double y, int* quo);  // see ??
float remquof(float x, float y, int* quo);
long double remquol(long double x, long double y, int* quo);

float copysign(float x, float y);  // see ??
double copysign(double x, double y);
long double copysign(long double x, long double y);  // see ??
float copysignf(float x, float y);
long double copysignl(long double x, long double y);

double nan(const char* tagp);
float nanf(const char* tagp);
long double nanl(const char* tagp);

float nextafter(float x, float y);  // see ??
double nextafter(double x, double y);
long double nextafter(long double x, long double y);  // see ??
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

float nexttoward(float x, long double y);  // see ??
double nexttoward(double x, long double y);
long double nexttoward(long double x, long double y);  // see ??
float nexttowardf(float x, long double y);
```

```
long double nexttowardl(long double x, long double y);

float fdim(float x, float y);  // see ??
double fdim(double x, double y);
long double fdim(long double x, long double y);  // see ??
float fdimf(float x, float y);
long double fdiml(long double x, long double y);

float fmax(float x, float y);  // see ??
double fmax(double x, double y);
long double fmax(long double x, long double y);  // see ??
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

float fmin(float x, float y);  // see ??
double fmin(double x, double y);
long double fmin(long double x, long double y);  // see ??
float fminf(float x, float y);
long double fminl(long double x, long double y);

float fma(float x, float y, float z);  // see ??
double fma(double x, double y, double z);
long double fma(long double x, long double y, long double z);  // see ??
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

// 26.8.4, linear interpolation
constexpr float lerp(float a, float b, float t);
constexpr double lerp(double a, double b, double t);
constexpr long double lerp(long double a, long double b, long double t);

// 26.8.5, classification / comparison functions
int fpclassify(float x);
int fpclassify(double x);
int fpclassify(long double x);

bool isfinite(float x);
bool isfinite(double x);
bool isfinite(long double x);

bool isinf(float x);
bool isinf(double x);
bool isinf(long double x);

bool isnan(float x);
bool isnan(double x);
bool isnan(long double x);

bool isnormal(float x);
bool isnormal(double x);
bool isnormal(long double x);

bool signbit(float x);
bool signbit(double x);
bool signbit(long double x);

bool isgreater(float x, float y);
bool isgreater(double x, double y);
bool isgreater(long double x, long double y);

bool isgreaterequal(float x, float y);
bool isgreaterequal(double x, double y);
bool isgreaterequal(long double x, long double y);
```

```
bool isless(float x, float y);
bool isless(double x, double y);
bool isless(long double x, long double y);

bool islessequal(float x, float y);
bool islessequal(double x, double y);
bool islessequal(long double x, long double y);

bool islessgreater(float x, float y);
bool islessgreater(double x, double y);
bool islessgreater(long double x, long double y);

bool isunordered(float x, float y);
bool isunordered(double x, double y);
bool isunordered(long double x, long double y);

// 26.8.6, mathematical special functions

// 26.8.6.1, associated Laguerre polynomials
double       assoc_laguerre(unsigned n, unsigned m, double x);
float        assoc_laguerref(unsigned n, unsigned m, float x);
long double  assoc_laguerrel(unsigned n, unsigned m, long double x);

// 26.8.6.2, associated Legendre functions
double       assoc_legendre(unsigned l, unsigned m, double x);
float        assoc_legendref(unsigned l, unsigned m, float x);
long double  assoc_legendrel(unsigned l, unsigned m, long double x);

// 26.8.6.3, beta function
double       beta(double x, double y);
float        betaf(float x, float y);
long double  betal(long double x, long double y);

// 26.8.6.4, complete elliptic integral of the first kind
double       comp_ellint_1(double k);
float        comp_ellint_1f(float k);
long double  comp_ellint_1l(long double k);

// 26.8.6.5, complete elliptic integral of the second kind
double       comp_ellint_2(double k);
float        comp_ellint_2f(float k);
long double  comp_ellint_2l(long double k);

// 26.8.6.6, complete elliptic integral of the third kind
double       comp_ellint_3(double k, double nu);
float        comp_ellint_3f(float k, float nu);
long double  comp_ellint_3l(long double k, long double nu);

// 26.8.6.7, regular modified cylindrical Bessel functions
double       cyl_bessel_i(double nu, double x);
float        cyl_bessel_if(float nu, float x);
long double  cyl_bessel_il(long double nu, long double x);

// 26.8.6.8, cylindrical Bessel functions of the first kind
double       cyl_bessel_j(double nu, double x);
float        cyl_bessel_jf(float nu, float x);
long double  cyl_bessel_jl(long double nu, long double x);

// 26.8.6.9, irregular modified cylindrical Bessel functions
double       cyl_bessel_k(double nu, double x);
float        cyl_bessel_kf(float nu, float x);
long double  cyl_bessel_kl(long double nu, long double x);
```

```
  // 26.8.6.10, cylindrical Neumann functions;
  // cylindrical Bessel functions of the second kind
  double       cyl_neumann(double nu, double x);
  float        cyl_neumannf(float nu, float x);
  long double  cyl_neumannl(long double nu, long double x);

  // 26.8.6.11, incomplete elliptic integral of the first kind
  double       ellint_1(double k, double phi);
  float        ellint_1f(float k, float phi);
  long double  ellint_1l(long double k, long double phi);

  // 26.8.6.12, incomplete elliptic integral of the second kind
  double       ellint_2(double k, double phi);
  float        ellint_2f(float k, float phi);
  long double  ellint_2l(long double k, long double phi);

  // 26.8.6.13, incomplete elliptic integral of the third kind
  double       ellint_3(double k, double nu, double phi);
  float        ellint_3f(float k, float nu, float phi);
  long double  ellint_3l(long double k, long double nu, long double phi);

  // 26.8.6.14, exponential integral
  double       expint(double x);
  float        expintf(float x);
  long double  expintl(long double x);

  // 26.8.6.15, Hermite polynomials
  double       hermite(unsigned n, double x);
  float        hermitef(unsigned n, float x);
  long double  hermitel(unsigned n, long double x);

  // 26.8.6.16, Laguerre polynomials
  double       laguerre(unsigned n, double x);
  float        laguerref(unsigned n, float x);
  long double  laguerrel(unsigned n, long double x);

  // 26.8.6.17, Legendre polynomials
  double       legendre(unsigned l, double x);
  float        legendref(unsigned l, float x);
  long double  legendrel(unsigned l, long double x);

  // 26.8.6.18, Riemann zeta function
  double       riemann_zeta(double x);
  float        riemann_zetaf(float x);
  long double  riemann_zetal(long double x);

  // 26.8.6.19, spherical Bessel functions of the first kind
  double       sph_bessel(unsigned n, double x);
  float        sph_besself(unsigned n, float x);
  long double  sph_bessell(unsigned n, long double x);

  // 26.8.6.20, spherical associated Legendre functions
  double       sph_legendre(unsigned l, unsigned m, double theta);
  float        sph_legendref(unsigned l, unsigned m, float theta);
  long double  sph_legendrel(unsigned l, unsigned m, long double theta);

  // 26.8.6.21, spherical Neumann functions;
  // spherical Bessel functions of the second kind
  double       sph_neumann(unsigned n, double x);
  float        sph_neumannf(unsigned n, float x);
  long double  sph_neumannl(unsigned n, long double x);
}
```

1 The contents and meaning of the header `<cmath>` are the same as the C standard library header `<math.h>`, with the addition of a three-dimensional hypotenuse function (26.8.3) and the mathematical special functions described in 26.8.6. [*Note*: Several functions have additional overloads in this document, but they have the same behavior as in the C standard library (??). — *end note*]

2 For each set of overloaded functions within `<cmath>`, with the exception of `abs`, there shall be additional overloads sufficient to ensure:

1. If any argument of arithmetic type corresponding to a `double` parameter has type `long double`, then all arguments of arithmetic type (??) corresponding to `double` parameters are effectively cast to `long double`.
2. Otherwise, if any argument of arithmetic type corresponding to a `double` parameter has type `double` or an integer type, then all arguments of arithmetic type corresponding to `double` parameters are effectively cast to `double`.
3. Otherwise, all arguments of arithmetic type corresponding to `double` parameters have type `float`.

[*Note*: `abs` is exempted from these rules in order to stay compatible with C. — *end note*]

SEE ALSO: ISO C 7.12

26.8.2 Absolute values [c.math.abs]

1 [*Note*: The headers `<cstdlib>` (??) and `<cmath>` (26.8.1) declare the functions described in this subclause. — *end note*]

```
int abs(int j);
long int abs(long int j);
long long int abs(long long int j);
float abs(float j);
double abs(double j);
long double abs(long double j);
```

2 *Effects:* The `abs` functions have the semantics specified in the C standard library for the functions `abs`, `labs`, `llabs`, `fabsf`, `fabs`, and `fabsl`.

3 *Remarks:* If `abs()` is called with an argument of type `X` for which `is_unsigned_v<X>` is `true` and if `X` cannot be converted to `int` by integral promotion (??), the program is ill-formed. [*Note*: Arguments that can be promoted to `int` are permitted for compatibility with C. — *end note*]

SEE ALSO: ISO C 7.12.7.2, 7.22.6.1

26.8.3 Three-dimensional hypotenuse [c.math.hypot3]

```
float hypot(float x, float y, float z);
double hypot(double x, double y, double z);
long double hypot(long double x, long double y, long double z);
```

1 *Returns:* $\sqrt{x^2 + y^2 + z^2}$.

26.8.4 Linear interpolation [c.math.lerp]

```
constexpr float lerp(float a, float b, float t);
constexpr double lerp(double a, double b, double t);
constexpr long double lerp(long double a, long double b, long double t);
```

1 *Returns:* $a + t(b - a)$.

2 *Remarks:* Let `r` be the value returned. If `isfinite(a) && isfinite(b)`, then:

(2.1) — If `t == 0`, then `r == a`.

(2.2) — If `t == 1`, then `r == b`.

(2.3) — If `t >= 0 && t <= 1`, then `isfinite(r)`.

(2.4) — If `isfinite(t) && a == b`, then `r == a`.

(2.5) — If `isfinite(t) || !isnan(t) && b-a != 0`, then `!isnan(r)`.

Let *`CMP`*`(x,y)` be `1` if `x > y`, `-1` if `x < y`, and `0` otherwise. For any `t1` and `t2`, the product of *`CMP`*`(lerp(a, b, t2), lerp(a, b, t1))`, *`CMP`*`(t2, t1)`, and *`CMP`*`(b, a)` is non-negative.

26.8.5 Classification / comparison functions [c.math.fpclass]

1 The classification / comparison functions behave the same as the C macros with the corresponding names defined in the C standard library. Each function is overloaded for the three floating-point types.

SEE ALSO: ISO C 7.12.3, 7.12.4

26.8.6 Mathematical special functions [sf.cmath]

1 If any argument value to any of the functions specified in this subclause is a NaN (Not a Number), the function shall return a NaN but it shall not report a domain error. Otherwise, the function shall report a domain error for just those argument values for which:

(1.1) — the function description's *Returns:* clause explicitly specifies a domain and those argument values fall outside the specified domain, or

(1.2) — the corresponding mathematical function value has a nonzero imaginary component, or

(1.3) — the corresponding mathematical function is not mathematically defined.[255]

2 Unless otherwise specified, each function is defined for all finite values, for negative infinity, and for positive infinity.

26.8.6.1 Associated Laguerre polynomials [sf.cmath.assoc.laguerre]

```
double       assoc_laguerre(unsigned n, unsigned m, double x);
float        assoc_laguerref(unsigned n, unsigned m, float x);
long double  assoc_laguerrel(unsigned n, unsigned m, long double x);
```

1 *Effects:* These functions compute the associated Laguerre polynomials of their respective arguments `n`, `m`, and `x`.

2 *Returns:*

$$\mathsf{L}_n^m(x) = (-1)^m \frac{\mathsf{d}^m}{\mathsf{d}x^m} \mathsf{L}_{n+m}(x) , \quad \text{for } x \geq 0,$$

where n is `n`, m is `m`, and x is `x`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `n >= 128` or if `m >= 128`.

26.8.6.2 Associated Legendre functions [sf.cmath.assoc.legendre]

```
double       assoc_legendre(unsigned l, unsigned m, double x);
float        assoc_legendref(unsigned l, unsigned m, float x);
long double  assoc_legendrel(unsigned l, unsigned m, long double x);
```

1 *Effects:* These functions compute the associated Legendre functions of their respective arguments `l`, `m`, and `x`.

2 *Returns:*

$$\mathsf{P}_\ell^m(x) = (1 - x^2)^{m/2} \frac{\mathsf{d}^m}{\mathsf{d}x^m} \mathsf{P}_\ell(x) , \quad \text{for } |x| \leq 1,$$

where l is `l`, m is `m`, and x is `x`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `l >= 128`.

26.8.6.3 Beta function [sf.cmath.beta]

```
double       beta(double x, double y);
float        betaf(float x, float y);
long double  betal(long double x, long double y);
```

1 *Effects:* These functions compute the beta function of their respective arguments `x` and `y`.

2 *Returns:*

$$\mathsf{B}(x, y) = \frac{\Gamma(x)\,\Gamma(y)}{\Gamma(x + y)} , \quad \text{for } x > 0,\ y > 0,$$

where x is `x` and y is `y`.

255) A mathematical function is mathematically defined for a given set of argument values (a) if it is explicitly defined for that set of argument values, or (b) if its limiting value exists and does not depend on the direction of approach.

26.8.6.4 Complete elliptic integral of the first kind [sf.cmath.comp.ellint.1]

```
double      comp_ellint_1(double k);
float       comp_ellint_1f(float k);
long double comp_ellint_1l(long double k);
```

1 *Effects:* These functions compute the complete elliptic integral of the first kind of their respective arguments k.

2 *Returns:*

$$\mathsf{K}(k) = \mathsf{F}(k, \pi/2)\ , \quad \text{for } |k| \le 1,$$

where k is k.

3 See also 26.8.6.11.

26.8.6.5 Complete elliptic integral of the second kind [sf.cmath.comp.ellint.2]

```
double      comp_ellint_2(double k);
float       comp_ellint_2f(float k);
long double comp_ellint_2l(long double k);
```

1 *Effects:* These functions compute the complete elliptic integral of the second kind of their respective arguments k.

2 *Returns:*

$$\mathsf{E}(k) = \mathsf{E}(k, \pi/2)\ , \quad \text{for } |k| \le 1,$$

where k is k.

3 See also 26.8.6.12.

26.8.6.6 Complete elliptic integral of the third kind [sf.cmath.comp.ellint.3]

```
double      comp_ellint_3(double k, double nu);
float       comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);
```

1 *Effects:* These functions compute the complete elliptic integral of the third kind of their respective arguments k and nu.

2 *Returns:*

$$\mathsf{\Pi}(\nu, k) = \mathsf{\Pi}(\nu, k, \pi/2)\ , \quad \text{for } |k| \le 1,$$

where k is k and ν is nu.

3 See also 26.8.6.13.

26.8.6.7 Regular modified cylindrical Bessel functions [sf.cmath.cyl.bessel.i]

```
double      cyl_bessel_i(double nu, double x);
float       cyl_bessel_if(float nu, float x);
long double cyl_bessel_il(long double nu, long double x);
```

1 *Effects:* These functions compute the regular modified cylindrical Bessel functions of their respective arguments nu and x.

2 *Returns:*

$$\mathsf{I}_\nu(x) = \mathrm{i}^{-\nu} \mathsf{J}_\nu(\mathrm{i}x) = \sum_{k=0}^{\infty} \frac{(x/2)^{\nu+2k}}{k!\,\Gamma(\nu+k+1)}\ , \quad \text{for } x \ge 0,$$

where ν is nu and x is x.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `nu >= 128`.

4 See also 26.8.6.8.

26.8.6.8 Cylindrical Bessel functions of the first kind [sf.cmath.cyl.bessel.j]

```
double      cyl_bessel_j(double nu, double x);
float       cyl_bessel_jf(float nu, float x);
long double cyl_bessel_jl(long double nu, long double x);
```

1 *Effects:* These functions compute the cylindrical Bessel functions of the first kind of their respective arguments `nu` and `x`.

2 *Returns:*

$$\mathsf{J}_\nu(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{\nu+2k}}{k!\,\Gamma(\nu+k+1)}\ , \quad \text{for } x \geq 0,$$

where ν is `nu` and x is `x`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `nu >= 128`.

26.8.6.9 Irregular modified cylindrical Bessel functions [sf.cmath.cyl.bessel.k]

```
double      cyl_bessel_k(double nu, double x);
float       cyl_bessel_kf(float nu, float x);
long double cyl_bessel_kl(long double nu, long double x);
```

1 *Effects:* These functions compute the irregular modified cylindrical Bessel functions of their respective arguments `nu` and `x`.

2 *Returns:*

$$\mathsf{K}_\nu(x) = (\pi/2)\mathrm{i}^{\nu+1}(\mathsf{J}_\nu(\mathrm{i}x) + \mathrm{i}\mathsf{N}_\nu(\mathrm{i}x)) = \begin{cases} \dfrac{\pi}{2}\dfrac{\mathsf{I}_{-\nu}(x) - \mathsf{I}_\nu(x)}{\sin\nu\pi}, & \text{for } x \geq 0 \text{ and non-integral } \nu \\[2ex] \dfrac{\pi}{2}\lim\limits_{\mu\to\nu}\dfrac{\mathsf{I}_{-\mu}(x) - \mathsf{I}_\mu(x)}{\sin\mu\pi}, & \text{for } x \geq 0 \text{ and integral } \nu \end{cases}$$

where ν is `nu` and x is `x`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `nu >= 128`.

4 See also 26.8.6.7, 26.8.6.8, 26.8.6.10.

26.8.6.10 Cylindrical Neumann functions [sf.cmath.cyl.neumann]

```
double      cyl_neumann(double nu, double x);
float       cyl_neumannf(float nu, float x);
long double cyl_neumannl(long double nu, long double x);
```

1 *Effects:* These functions compute the cylindrical Neumann functions, also known as the cylindrical Bessel functions of the second kind, of their respective arguments `nu` and `x`.

2 *Returns:*

$$\mathsf{N}_\nu(x) = \begin{cases} \dfrac{\mathsf{J}_\nu(x)\cos\nu\pi - \mathsf{J}_{-\nu}(x)}{\sin\nu\pi}, & \text{for } x \geq 0 \text{ and non-integral } \nu \\[2ex] \lim\limits_{\mu\to\nu}\dfrac{\mathsf{J}_\mu(x)\cos\mu\pi - \mathsf{J}_{-\mu}(x)}{\sin\mu\pi}, & \text{for } x \geq 0 \text{ and integral } \nu \end{cases}$$

where ν is `nu` and x is `x`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `nu >= 128`.

4 See also 26.8.6.8.

26.8.6.11 Incomplete elliptic integral of the first kind [sf.cmath.ellint.1]

```
double      ellint_1(double k, double phi);
float       ellint_1f(float k, float phi);
long double ellint_1l(long double k, long double phi);
```

1 *Effects:* These functions compute the incomplete elliptic integral of the first kind of their respective arguments `k` and `phi` (`phi` measured in radians).

2 *Returns:*

$$\mathsf{F}(k, \phi) = \int_0^\phi \frac{\mathsf{d}\theta}{\sqrt{1 - k^2 \sin^2 \theta}} \ , \quad \text{for } |k| \le 1,$$

where k is `k` and ϕ is `phi`.

26.8.6.12 Incomplete elliptic integral of the second kind [sf.cmath.ellint.2]

```
double       ellint_2(double k, double phi);
float        ellint_2f(float k, float phi);
long double  ellint_2l(long double k, long double phi);
```

1 *Effects:* These functions compute the incomplete elliptic integral of the second kind of their respective arguments `k` and `phi` (`phi` measured in radians).

2 *Returns:*

$$\mathsf{E}(k, \phi) = \int_0^\phi \sqrt{1 - k^2 \sin^2 \theta}\, \mathsf{d}\theta \ , \quad \text{for } |k| \le 1,$$

where k is `k` and ϕ is `phi`.

26.8.6.13 Incomplete elliptic integral of the third kind [sf.cmath.ellint.3]

```
double       ellint_3(double k, double nu, double phi);
float        ellint_3f(float k, float nu, float phi);
long double  ellint_3l(long double k, long double nu, long double phi);
```

1 *Effects:* These functions compute the incomplete elliptic integral of the third kind of their respective arguments `k`, `nu`, and `phi` (`phi` measured in radians).

2 *Returns:*

$$\mathsf{\Pi}(\nu, k, \phi) = \int_0^\phi \frac{\mathsf{d}\theta}{(1 - \nu \ \sin^2 \theta)\sqrt{1 - k^2 \sin^2 \theta}} \ , \quad \text{for } |k| \le 1,$$

where ν is `nu`, k is `k`, and ϕ is `phi`.

26.8.6.14 Exponential integral [sf.cmath.expint]

```
double       expint(double x);
float        expintf(float x);
long double  expintl(long double x);
```

1 *Effects:* These functions compute the exponential integral of their respective arguments `x`.

2 *Returns:*

$$\mathsf{Ei}(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t} \, \mathsf{d}t$$

where x is `x`.

26.8.6.15 Hermite polynomials [sf.cmath.hermite]

```
double       hermite(unsigned n, double x);
float        hermitef(unsigned n, float x);
long double  hermitel(unsigned n, long double x);
```

1 *Effects:* These functions compute the Hermite polynomials of their respective arguments `n` and `x`.

2 *Returns:*

$$\mathsf{H}_n(x) = (-1)^n e^{x^2} \frac{\mathsf{d}^n}{\mathsf{d}x^n} \, e^{-x^2}$$

where n is `n` and x is `x`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `n >= 128`.

26.8.6.16 Laguerre polynomials [sf.cmath.laguerre]

```
double      laguerre(unsigned n, double x);
float       laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);
```

1 *Effects:* These functions compute the Laguerre polynomials of their respective arguments `n` and `x`.

2 *Returns:*

$$\mathsf{L}_n(x) = \frac{e^x}{n!} \frac{\mathsf{d}^n}{\mathsf{d}x^n} \left(x^n e^{-x}\right) , \quad \text{for } x \geq 0,$$

where n is `n` and x is `x`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `n >= 128`.

26.8.6.17 Legendre polynomials [sf.cmath.legendre]

```
double      legendre(unsigned l, double x);
float       legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);
```

1 *Effects:* These functions compute the Legendre polynomials of their respective arguments `l` and `x`.

2 *Returns:*

$$\mathsf{P}_\ell(x) = \frac{1}{2^\ell\, \ell!} \frac{\mathsf{d}^\ell}{\mathsf{d}x^\ell} \left(x^2 - 1\right)^\ell , \quad \text{for } |x| \leq 1,$$

where l is `l` and x is `x`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `l >= 128`.

26.8.6.18 Riemann zeta function [sf.cmath.riemann.zeta]

```
double      riemann_zeta(double x);
float       riemann_zetaf(float x);
long double riemann_zetal(long double x);
```

1 *Effects:* These functions compute the Riemann zeta function of their respective arguments `x`.

2 *Returns:*

$$\zeta(x) = \begin{cases} \displaystyle\sum_{k=1}^{\infty} k^{-x}, & \text{for } x > 1 \\ \displaystyle\frac{1}{1-2^{1-x}} \sum_{k=1}^{\infty} (-1)^{k-1} k^{-x}, & \text{for } 0 \leq x \leq 1 \\ 2^x \pi^{x-1} \sin(\frac{\pi x}{2})\, \Gamma(1-x)\, \zeta(1-x), & \text{for } x < 0 \end{cases}$$

where x is `x`.

26.8.6.19 Spherical Bessel functions of the first kind [sf.cmath.sph.bessel]

```
double      sph_bessel(unsigned n, double x);
float       sph_besself(unsigned n, float x);
long double sph_bessell(unsigned n, long double x);
```

1 *Effects:* These functions compute the spherical Bessel functions of the first kind of their respective arguments `n` and `x`.

2 *Returns:*

$$\mathsf{j}_n(x) = (\pi/2x)^{1\!/\!2} \mathsf{J}_{n+1\!/\!2}(x) , \quad \text{for } x \geq 0,$$

where n is `n` and x is `x`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `n >= 128`.

4 See also 26.8.6.8.

26.8.6.20 Spherical associated Legendre functions [sf.cmath.sph.legendre]

```
double      sph_legendre(unsigned l, unsigned m, double theta);
float       sph_legendref(unsigned l, unsigned m, float theta);
long double sph_legendrel(unsigned l, unsigned m, long double theta);
```

1 *Effects:* These functions compute the spherical associated Legendre functions of their respective arguments `l`, `m`, and `theta` (`theta` measured in radians).

2 *Returns:*

$$\mathsf{Y}_\ell^m(\theta, 0)$$

where

$$\mathsf{Y}_\ell^m(\theta,\phi) = (-1)^m \left[\frac{(2\ell+1)}{4\pi}\frac{(\ell-m)!}{(\ell+m)!}\right]^{1/2} \mathsf{P}_\ell^m(\cos\theta)e^{im\phi}\ , \quad \text{for } |m| \le \ell,$$

and l is `l`, m is `m`, and θ is `theta`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `l >= 128`.

4 See also 26.8.6.2.

26.8.6.21 Spherical Neumann functions [sf.cmath.sph.neumann]

```
double      sph_neumann(unsigned n, double x);
float       sph_neumannf(unsigned n, float x);
long double sph_neumannl(unsigned n, long double x);
```

1 *Effects:* These functions compute the spherical Neumann functions, also known as the spherical Bessel functions of the second kind, of their respective arguments `n` and `x`.

2 *Returns:*

$$\mathsf{n}_n(x) = (\pi/2x)^{1/2}\mathsf{N}_{n+1/2}(x)\ , \quad \text{for } x \ge 0,$$

where n is `n` and x is `x`.

3 *Remarks:* The effect of calling each of these functions is implementation-defined if `n >= 128`.

4 See also 26.8.6.10.

26.9 Numbers [numbers]

26.9.1 Header <numbers> synopsis [numbers.syn]

```
namespace std::numbers {
  template<class T> inline constexpr T e_v          = unspecified;
  template<class T> inline constexpr T log2e_v      = unspecified;
  template<class T> inline constexpr T log10e_v     = unspecified;
  template<class T> inline constexpr T pi_v         = unspecified;
  template<class T> inline constexpr T inv_pi_v     = unspecified;
  template<class T> inline constexpr T inv_sqrtpi_v = unspecified;
  template<class T> inline constexpr T ln2_v        = unspecified;
  template<class T> inline constexpr T ln10_v       = unspecified;
  template<class T> inline constexpr T sqrt2_v      = unspecified;
  template<class T> inline constexpr T sqrt3_v      = unspecified;
  template<class T> inline constexpr T inv_sqrt3_v  = unspecified;
  template<class T> inline constexpr T egamma_v     = unspecified;
  template<class T> inline constexpr T phi_v        = unspecified;

  template<floating_point T> inline constexpr T e_v<T>          = see below;
  template<floating_point T> inline constexpr T log2e_v<T>      = see below;
  template<floating_point T> inline constexpr T log10e_v<T>     = see below;
  template<floating_point T> inline constexpr T pi_v<T>         = see below;
  template<floating_point T> inline constexpr T inv_pi_v<T>     = see below;
  template<floating_point T> inline constexpr T inv_sqrtpi_v<T> = see below;
  template<floating_point T> inline constexpr T ln2_v<T>        = see below;
  template<floating_point T> inline constexpr T ln10_v<T>       = see below;
  template<floating_point T> inline constexpr T sqrt2_v<T>      = see below;
  template<floating_point T> inline constexpr T sqrt3_v<T>      = see below;
  template<floating_point T> inline constexpr T inv_sqrt3_v<T>  = see below;
  template<floating_point T> inline constexpr T egamma_v<T>     = see below;
```

```
  template<floating_point T> inline constexpr T phi_v<T>        = see below;

  inline constexpr double e          = e_v<double>;
  inline constexpr double log2e      = log2e_v<double>;
  inline constexpr double log10e     = log10e_v<double>;
  inline constexpr double pi         = pi_v<double>;
  inline constexpr double inv_pi     = inv_pi_v<double>;
  inline constexpr double inv_sqrtpi = inv_sqrtpi_v<double>;
  inline constexpr double ln2        = ln2_v<double>;
  inline constexpr double ln10       = ln10_v<double>;
  inline constexpr double sqrt2      = sqrt2_v<double>;
  inline constexpr double sqrt3      = sqrt3_v<double>;
  inline constexpr double inv_sqrt3  = inv_sqrt3_v<double>;
  inline constexpr double egamma     = egamma_v<double>;
  inline constexpr double phi        = phi_v<double>;
}
```

26.9.2 Mathematical constants [math.constants]

1 The library-defined partial specializations of mathematical constant variable templates are initialized with the nearest representable values of e, $\log_2 e$, $\log_{10} e$, π, $\frac{1}{\pi}$, $\frac{1}{\sqrt{\pi}}$, $\ln 2$, $\ln 10$, $\sqrt{2}$, $\sqrt{3}$, $\frac{1}{\sqrt{3}}$, the Euler-Mascheroni γ constant, and the golden ratio ϕ constant $\frac{1+\sqrt{5}}{2}$, respectively.

2 Pursuant to **??**, a program may partially or explicitly specialize a mathematical constant variable template provided that the specialization depends on a program-defined type.

3 A program that instantiates a primary template of a mathematical constant variable template is ill-formed.