

# Concepts for the C++0x Standard Library: Numerics (Revision 4)

Douglas Gregor, and Andrew Lumsdaine  
Open Systems Laboratory  
Indiana University  
Bloomington, IN 47405  
{[dgregor](mailto:dgregor@osl.iu.edu), [lums](mailto:lums@osl.iu.edu)}@osl.iu.edu

Document number: DRAFT  
Revises document number: N2736=08-0246  
Date: 2008-09-12  
Project: Programming Language C++, Library Working Group  
Reply-to: Douglas Gregor <[dgregor@osl.iu.edu](mailto:dgregor@osl.iu.edu)>

## Introduction

This document proposes changes to Chapter 26 of the C++ Standard Library in order to make full use of concepts [1]. We make every attempt to provide complete backward compatibility with the pre-concept Standard Library, and note each place where we have knowingly changed semantics.

This document is formatted in the same manner as the latest working draft of the C++ standard (N2691). Future versions of this document will track the working draft and the concepts proposal as they evolve. Wherever the numbering of a (sub)section matches a section of the working paper, the text in this document should be considered replacement text, unless editorial comments state otherwise. All editorial comments will **have a gray background**. Changes to the replacement text are categorized and typeset as additions, ~~removals~~, or ~~changes~~modifications.

## Changes from N2736

- Finished description of HasAcons.
- Clean up Has\* requirements in valarray.
- Added DefaultConstructible requirement to valarray.
- Synchronized the complex numbers section with the latest C++0x working paper.

**Issues resolved in this paper**

The following issue has been applied to this paper and should be resolved as NAD:

- Issue 844: complex pow return type is ambiguous. Rather than add concepts to the `pow(complex, int)` function, which will be removed by the proposed resolution to issue 844, we have applied the proposed resolution to this document.

---

---

# Chapter 26 Numerics library

[**numerics**]

---

---

- 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 facilities included from the ISO C library, as summarized in Table 1.

Table 1: Numerics library summary

Subclause	Header(s)
<a href="#">26.1 Requirements</a> <a href="#">Concepts</a>	<code>&lt;numeric_concepts&gt;</code>
<a href="#">26.3</a> Complex Numbers	<code>&lt;complex&gt;</code>
<a href="#">??</a> Random number generation	<code>&lt;random&gt;</code>
<a href="#">26.5</a> Numeric arrays	<code>&lt;valarray&gt;</code>
<a href="#">26.6</a> Generalized numeric operations	<code>&lt;numeric&gt;</code>
<a href="#">??</a> C library	<code>&lt;cmath&gt;</code> <code>&lt;ctgmath&gt;</code> <code>&lt;tgmath.h&gt;</code> <code>&lt;cstdlib&gt;</code>

## 26.1 Numeric type concepts

[**numeric.concepts**]

The name of this section has been changed from “Numeric type requirements” to “Numeric type concepts”, and the label has been changed from [numeric.requirements] to [numeric.concepts].

- 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 type `T` that satisfies the [following requirements](#): [Semiregular concept](#) ([??](#)).<sup>1)</sup>
  - `T` is not an abstract class (it has no pure virtual member functions);
  - `T` is not a reference type;
  - `T` is not cv-qualified;
  - If `T` is a class, it has a public default constructor;

---

<sup>1)</sup> In other words, value types. These include built-in arithmetic types, pointers, the library class `complex`, and instantiations of `valarray` for value types.

- If  $T$  is a class, it has a public copy constructor with the signature  $T::T(\text{const } T\&)$
- If  $T$  is a class, it has a public destructor;
- If  $T$  is a class, it has a public assignment operator whose signature is either  $T\& T::\text{operator}=(\text{const } T\&)$  or  $T\& T::\text{operator}=(T)$
- [*Note:* This rule states that there shall not be any subtle differences in the semantics of initialization versus assignment. This gives an implementation considerable flexibility in how arrays are initialized.

[*Example:* An implementation is allowed to initialize a `valarray` by allocating storage using the `new` operator (which implies a call to the default constructor for each element) and then assigning each element its value. Or the implementation can allocate raw storage and use the copy constructor to initialize each element. — *end example*]

If the distinction between initialization and assignment is important for a class, or if it fails to satisfy any of the other conditions listed above, the programmer should use `vector` (??) instead of `valarray` for that class; — *end note*]

- If  $T$  is a class, it does not overload unary operator `&`.

- 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  $T$  satisfies 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*]

#### Header `<numeric_concepts>` synopsis

```
namespace std {
    auto concept HasAbs<typename T> see below;
    auto concept HasAcos<typename T> see below;
    auto concept HasAsin<typename T> see below;
    auto concept HasAtan<typename T> see below;
    auto concept HasAtan2<typename T> see below;
    auto concept HasCos<typename T> see below;
    auto concept HasCosh<typename T> see below;
    auto concept HasExp<typename T> see below;
    auto concept HasLog<typename T> see below;
    auto concept HasLog10<typename T> see below;
    auto concept HasPow<typename T> see below;
    auto concept HasSin<typename T> see below;
    auto concept HasSinh<typename T> see below;
    auto concept HasSqrt<typename T> see below;
    auto concept HasTan<typename T> see below;
    auto concept HasTanh<typename T> see below;
}
```

These transcendental concepts are here to support the transcendental operations of `valarray` and `complex`. They are separate concepts—rather than a single Transcendental concept—because the `valarray` versions in C++03 specifically require only the corresponding operation.

```
auto concept HasAbs<typename T> {  
    T abs(const T&);  
}
```

5 *Note:* describes types for which an absolute value can be computed.

```
auto concept HasAcos<typename T> {  
    T acos(const T&);  
}
```

6 *Note:* describes types for which an arc cosine can be computed.

```
auto concept HasAsin<typename T> {  
    T asin(const T&);  
}
```

7 *Note:* describes types for which an arc sine can be computed.

```
auto concept HasAtan<typename T> {  
    T atan(const T&);  
}
```

8 *Note:* describes types for which an arc tangent can be computed.

```
auto concept HasAtan2<typename T> {  
    T atan2(const T& y, const T& x);  
}
```

9 *Note:* describes types for which an arc tangent can be computed from  $y/x$ .

```
auto concept HasCos<typename T> {  
    T cos(const T&);  
}
```

10 *Note:* describes types for which a cosine can be computed.

```
auto concept HasCosh<typename T> {  
    T cosh(const T&);  
}
```

11 *Note:* describes types for which a hyperbolic cosine can be computed.

```
auto concept HasExp<typename T> {  
    T exp(const T&);  
}
```

12 *Note:* describes types for which the base- $e$  exponential can be computed.

```
auto concept HasLog<typename T> {  
    T log(const T&);  
}
```

13 *Note:* describes types for which the natural logarithm can be computed.

```
auto concept HasLog10<typename T> {
    T log10(const T&);
}
```

14 Note: describes types for which a base-10 logarithm can be computed.

```
auto concept HasPow<typename T> {
    T pow(const T& x, const T& y);
}
```

15 Note: describes types for which the exponential  $x^y$  can be computed.

```
auto concept HasSin<typename T> {
    T sin(const T&);
}
```

16 Note: describes types for which the sine can be computed.

```
auto concept HasSinh<typename T> {
    T sinh(const T&);
}
```

17 Note: describes types for which the hyperbolic sine can be computed.

```
auto concept HasSqrt<typename T> {
    T sqrt(const T&);
}
```

18 Note: describes types for which the square root can be computed.

```
auto concept HasTan<typename T> {
    T tan(const T&);
}
```

19 Note: describes types for which the tangent can be computed.

```
auto concept HasTanh<typename T> {
    T tanh(const T&);
}
```

20 Note: describes types for which the hyperbolic tangent can be computed.

### 26.3 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.

### 26.3.1 Header `<complex>` synopsis

[[complex.synopsis](#)]

```

namespace std {
    template<class ArithmeticLike T> class complex;
    template<> class complex<float>;
    template<> class complex<double>;
    template<> class complex<long double>;

    // 26.3.6 operators:
    template<class ArithmeticLike T>
        complex<T> operator+(const complex<T>&, const complex<T>&);
    template<class ArithmeticLike T> complex<T> operator+(const complex<T>&, const T&);
    template<class ArithmeticLike T> complex<T> operator+(const T&, const complex<T>&);

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

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

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

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

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

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

```

```

// 26.3.7 values:
template<class ArithmeticLike T> T real(const complex<T>&);
template<class ArithmeticLike T> T imag(const complex<T>&);

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

template<class ArithmeticLike T> complex<T> conj(const complex<T>&);
template<class FloatingPointType T> complex<T> proj(const complex<T>&);
template<class FloatingPointType T> complex<T> polar(const T&, const T& = 0);

// 26.3.8 transcendentals:
template<class FloatingPointType T> complex<T> acos(const complex<T>&);
template<class FloatingPointType T> complex<T> asin(const complex<T>&);
template<class FloatingPointType T> complex<T> atan(const complex<T>&);

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

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

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

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

```

### 26.3.2 Class template `complex`

[complex]

```

namespace std {
  template<class ArithmeticLike T>
  class complex {
  public:
    typedef T value_type;

    complex(const T& re = T(), const T& im = T());
    complex(const complex&);

```

Draft

```

template<class ArithmeticLike X> requires Constructible<T, X>
    complex(const complex<X>&);

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

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

complex& operator=(const complex&);
template<class ArithmeticLike X> requires Convertible<X, T>
    complex<T>& operator= (const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, T>
    complex<T>& operator+=(const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, T>
    complex<T>& operator-=(const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, T>
    complex<T>& operator*=(const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, T>
    complex<T>& operator/=(const complex<X>&);
};

}

```

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

### 26.3.3 complex specializations

[`complex.special`]

```

template<> class complex<float> {
public:
    typedef float value_type;

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

    float real() const;
    float imag() const;

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

```

```

complex<float>& operator=(const complex<float>&);
template<class ArithmeticLike X> requires Convertible<X, float>
    complex<float>& operator= (const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, float>
    complex<float>& operator+=(const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, float>
    complex<float>& operator-=(const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, float>
    complex<float>& operator*=(const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, float>
    complex<float>& operator/=(const complex<X>&);
};

template<> class complex<double> {
public:
    typedef double value_type;

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

    double real() const;
    double imag() const;

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

    complex<double>& operator=(const complex<double>&);
    template<class ArithmeticLike X> requires Convertible<X, double>
        complex<double>& operator= (const complex<X>&);
    template<class ArithmeticLike X> requires Convertible<X, double>
        complex<double>& operator+=(const complex<X>&);
    template<class ArithmeticLike X> requires Convertible<X, double>
        complex<double>& operator-=(const complex<X>&);
    template<class ArithmeticLike X> requires Convertible<X, double>
        complex<double>& operator*=(const complex<X>&);
    template<class ArithmeticLike X> requires Convertible<X, double>
        complex<double>& operator/=(const complex<X>&);
};

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

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

```

```

complex(const complex<double>&);

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

complex<long double>& operator=(const complex<long double>&);
complex<long double>& operator= (long double);
complex<long double>& operator+=(long double);
complex<long double>& operator-=(long double);
complex<long double>& operator*=(long double);
complex<long double>& operator/=(long double);

template<class ArithmeticLike X> requires Convertible<X, long double>
complex<long double>& operator= (const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, long double>
complex<long double>& operator+=(const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, long double>
complex<long double>& operator-=(const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, long double>
complex<long double>& operator*=(const complex<X>&);
template<class ArithmeticLike X> requires Convertible<X, long double>
complex<long double>& operator/=(const complex<X>&);
};

```

**26.3.4 complex member functions****[complex.members]****26.3.5 complex member operators****[complex.member.ops]**

```
complex<T>& 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*.

```
complex<T>& 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*.

```
complex<T>& 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*.

```
complex<T>& 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*.

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

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

10 *Returns:* *\*this*.

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

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

12 *Returns:* *\*this*.

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

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

*Returns:* *\*this*.

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

14 *Effects:* Divides the complex value *rhs* into the complex value *\*this* and stores the quotient in *\*this*.

15 *Returns:* *\*this*.

### 26.3.6 complex non-member operations

[complex.ops]

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

1 *Remarks:* unary operator.

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

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

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

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

4 *Remarks:* unary operator.

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

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

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

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

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

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

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

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

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

11 *Returns:* `rhs.real() != lhs.real() || rhs.imag() != lhs.imag()`.

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

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 *Requires:* The input values be convertible to `T`.

If bad input is encountered, calls `is.setstate(ios::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:

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

```

### 26.3.7 complex value operations

[complex.value.ops]

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

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

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

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

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

3 *Returns:* the magnitude of  $x$ .

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

4 *Returns:* the phase angle of  $x$ , or  $\text{atan2}(\text{imag}(x), \text{real}(x))$ .

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

5 *Returns:* the squared magnitude of  $x$ .

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

6 *Returns:* the complex conjugate of  $x$ .

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

7 *Effects:* Behaves the same as C99 function `cproj`, defined in subclause 7.3.9.4.

```
template<class FloatingPointType T> complex<T> polar(const T& rho, const T& theta = 0);
```

8 *Returns:* the complex value corresponding to a complex number whose magnitude is  $\rho$  and whose phase angle is  $\theta$ .

### 26.3.8 complex transcendentals

[complex.transcendentals]

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

1 *Effects:* Behaves the same as C99 function `caacos`, defined in subclause 7.3.5.1.

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

2 *Effects:* Behaves the same as C99 function `casin`, defined in subclause 7.3.5.2.

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

- 3 *Effects:* Behaves the same as C99 function `catan`, defined in subclause 7.3.5.3.
- ```
template<class FloatingPointType T> complex<T> acosh(const complex<T>& x);
```
- 4 *Effects:* Behaves the same as C99 function `cacosh`, defined in subclause 7.3.6.1.
- ```
template<class FloatingPointType T> complex<T> asinh(const complex<T>& x);
```
- 5 *Effects:* Behaves the same as C99 function `casinh`, defined in subclause 7.3.6.2.
- ```
template<class FloatingPointType T> complex<T> atanh(const complex<T>& x);
```
- 6 *Effects:* Behaves the same as C99 function `catanh`, defined in subclause 7.3.6.3.
- ```
template<class FloatingPointType T> complex<T> cos(const complex<T>& x);
```
- 7 *Returns:* the complex cosine of  $x$ .
- ```
template<class FloatingPointType T> complex<T> cosh(const complex<T>& x);
```
- 8 *Returns:* the complex hyperbolic cosine of  $x$ .
- ```
template<class FloatingPointType T> complex<T> exp(const complex<T>& x);
```
- 9 *Returns:* the complex base  $e$  exponential of  $x$ .
- ```
template<class FloatingPointType T> complex<T> log(const complex<T>& x);
```
- 10 *Remarks:* the branch cuts are along the negative real axis.
- 11 *Returns:* the complex natural (base  $e$ ) logarithm of  $x$ , in the range of a strip mathematically unbounded along the real axis and in the interval  $[-i \text{ times } \pi, i \text{ times } \pi]$  along the imaginary axis. When  $x$  is a negative real number, `imag(log(x))` is  $\pi$ .
- ```
template<class FloatingPointType T> complex<T> log10(const complex<T>& x);
```
- 12 *Remarks:* the branch cuts are along the negative real axis.
- 13 *Returns:* the complex common (base 10) logarithm of  $x$ , defined as  $\log(x)/\log(10)$ .
- ```
template<class T> complex<T> pow(const complex<T>& x, int y);
template<class FloatingPointType T> complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class FloatingPointType T> complex<T> pow(const complex<T>& x, const T& y);
template<class FloatingPointType T> complex<T> pow (const T& x, const complex<T>& y);
```
- 14 *Remarks:* the branch cuts are along the negative real axis.
- 15 *Returns:* the complex power of base  $x$  raised to the  $y$ -th power, defined as  $\exp(y * \log(x))$ . The value returned for `pow(0,0)` is implementation-defined.
- ```
template<class FloatingPointType T> complex<T> sin (const complex<T>& x);
```
- 16 *Returns:* the complex sine of  $x$ .
- ```
template<class FloatingPointType T> complex<T> sinh (const complex<T>& x);
```

17 *Returns:* the complex hyperbolic sine of  $x$ .

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

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

19 *Returns:* the complex square root of  $x$ , in the range of the right half-plane. If the argument is a negative real number, the value returned lies on the positive imaginary axis.

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

20 *Returns:* the complex tangent of  $x$ .

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

21 *Returns:* the complex hyperbolic tangent of  $x$ .

### 26.3.9 Additional Overloads

[**cmplx.over**]

1 The following function templates shall have additional overloads:

|                   |                    |
|-------------------|--------------------|
| <code>arg</code>  | <code>norm</code>  |
| <code>conj</code> | <code>polar</code> |
| <code>imag</code> | <code>real</code>  |

2 The additional overloads shall be sufficient to ensure:

1. If the argument has type `long double`, then it is effectively cast to `complex<long double>`.
2. Otherwise, if the argument has type `double` or an integer type, then it is effectively cast to `complex<double>`.
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>`:

1. If either argument has type `complex<long double>` or type `long double`, then both arguments are effectively cast to `complex<long double>`.
2. Otherwise, if either argument has type `complex<double>`, `double`, or an integer type, then both arguments are effectively cast to `complex<double>`.
3. Otherwise, if either argument has type `complex<float>` or `float`, then both arguments are effectively cast to `complex<float>`.

### 26.3.10 Header `<ccomplex>`

[**ccmplx**]

1 The header behaves as if it simply includes the header `<complex>`.

### 26.3.11 Header `<complex.h>`

[**cmplxh**]

1 The header behaves as if it includes the header `<ccomplex>`, and provides sufficient *using* declarations to declare in the

global namespace all function and type names declared or defined in the header `<complex>`.

## 26.5 Numeric arrays

[numarray]

### 26.5.1 Header `<valarray>` synopsis

[valarray.synopsis]

```

namespace std {
    template<class Semiregular T> requires DefaultConstructible<T>
        class valarray;           // An array of type T
    class slice;                  // a BLAS-like slice out of an array
    template<class Semiregular T> class slice_array;
    class gslice;                // a generalized slice out of an array
    template<class Semiregular T> class gslice_array;
    template<class Semiregular T> class mask_array;           // a masked array
    template<class Semiregular T> class indirect_array;      // an indirected array

    template<class Semiregular T> void swap(valarray<T>&, valarray<T>&);
    template<class Semiregular T> void swap(valarray<T>&&, valarray<T>&);
    template<class Semiregular T> void swap(valarray<T>&, valarray<T>&&);

    template<class T>
        requires HasMultiply<T, T> && Convertible<HasMultiply<T, T>::result_type, T>
        valarray<T> operator* (const valarray<T>&, const valarray<T>&);
    template<class T>
        requires HasMultiply<T, T> && Convertible<HasMultiply<T, T>::result_type, T>
        valarray<T> operator* (const valarray<T>&, const T&);
    template<class T>
        requires HasMultiply<T, T> && Convertible<HasMultiply<T, T>::result_type, T>
        valarray<T> operator* (const T&, const valarray<T>&);

    template<class T>
        requires HasDivide<T, T> && Convertible<HasDivide<T, T>::result_type, T>
        valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
    template<class T>
        requires HasDivide<T, T> && Convertible<HasDivide<T, T>::result_type, T>
        valarray<T> operator/ (const valarray<T>&, const T&);
    template<class T>
        requires HasDivide<T, T> && Convertible<HasDivide<T, T>::result_type, T>
        valarray<T> operator/ (const T&, const valarray<T>&);

    template<class T>
        requires HasModulus<T, T> && Convertible<HasModulus<T, T>::result_type, T>
        valarray<T> operator% (const valarray<T>&, const valarray<T>&);
    template<class T>
        requires HasModulus<T, T> && Convertible<HasModulus<T, T>::result_type, T>
        valarray<T> operator% (const valarray<T>&, const T&);
    template<class T>
        requires HasModulus<T, T> && Convertible<HasModulus<T, T>::result_type, T>
        valarray<T> operator% (const T&, const valarray<T>&);

    template<class T>

```

```

    requires HasPlus<T, T> && Convertible<HasPlus<T, T>::result_type, T>
    valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasPlus<T, T> && Convertible<HasPlus<T, T>::result_type, T>
    valarray<T> operator+ (const valarray<T>&, const T&);
template<class T>
    requires HasPlus<T, T> && Convertible<HasPlus<T, T>::result_type, T>
    valarray<T> operator+ (const T&, const valarray<T>&);

template<class T>
    requires HasMinus<T, T> && Convertible<HasMinus<T, T>::result_type, T>
    valarray<T> operator- (const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasMinus<T, T> && Convertible<HasMinus<T, T>::result_type, T>
    valarray<T> operator- (const valarray<T>&, const T&);
template<class T>
    requires HasMinus<T, T> && Convertible<HasMinus<T, T>::result_type, T>
    valarray<T> operator- (const T&, const valarray<T>&);

template<class T>
    requires HasBitXor<T, T> && Convertible<HasBitXor<T, T>::result_type, T>
    valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasBitXor<T, T> && Convertible<HasBitXor<T, T>::result_type, T>
    valarray<T> operator^ (const valarray<T>&, const T&);
template<class T>
    requires HasBitXor<T, T> && Convertible<HasBitXor<T, T>::result_type, T>
    valarray<T> operator^ (const T&, const valarray<T>&);

template<class T>
    requires HasBitAnd<T, T> && Convertible<HasBitAnd<T, T>::result_type, T>
    valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasBitAnd<T, T> && Convertible<HasBitAnd<T, T>::result_type, T>
    valarray<T> operator& (const valarray<T>&, const T&);
template<class T>
    requires HasBitAnd<T, T> && Convertible<HasBitAnd<T, T>::result_type, T>
    valarray<T> operator& (const T&, const valarray<T>&);

template<class T>
    requires HasBitOr<T, T> && Convertible<HasBitOr<T, T>::result_type, T>
    valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasBitOr<T, T> && Convertible<HasBitOr<T, T>::result_type, T>
    valarray<T> operator| (const valarray<T>&, const T&);
template<class T>
    requires HasBitOr<T, T> && Convertible<HasBitOr<T, T>::result_type, T>
    valarray<T> operator| (const T&, const valarray<T>&);

template<class T>

```

```

    requires HasLeftShift<T, T> && Convertible<HasLeftShift<T, T>::result_type, T>
    valarray<T> operator<<(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasLeftShift<T, T> && Convertible<HasLeftShift<T, T>::result_type, T>
    valarray<T> operator<<(const valarray<T>&, const T&);
template<class T>
    requires HasLeftShift<T, T> && Convertible<HasLeftShift<T, T>::result_type, T>
    valarray<T> operator<<(const T&, const valarray<T>&);

template<class T>
    requires HasRightShift<T, T> && Convertible<HasRightShift<T, T>::result_type, T>
    valarray<T> operator>>(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasRightShift<T, T> && Convertible<HasRightShift<T, T>::result_type, T>
    valarray<T> operator>>(const valarray<T>&, const T&);
template<class T>
    requires HasRightShift<T, T> && Convertible<HasRightShift<T, T>::result_type, T>
    valarray<T> operator>>(const T&, const valarray<T>&);

template<class T>
    requires HasLogicalAnd<T, T>
    valarray<bool> operator&&(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasLogicalAnd<T, T>
    valarray<bool> operator&&(const valarray<T>&, const T&);
template<class T>
    requires HasLogicalAnd<T, T>
    valarray<bool> operator&&(const T&, const valarray<T>&);

template<class T>
    requires HasLogicalOr<T, T>
    valarray<bool> operator|| (const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasLogicalOr<T, T>
    valarray<bool> operator|| (const valarray<T>&, const T&);
template<class T>
    requires HasLogicalOr<T, T>
    valarray<bool> operator|| (const T&, const valarray<T>&);

template<class T>
    requires EqualityComparable<T>
    valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
template<class T>
    requires EqualityComparable<T>
    valarray<bool> operator==(const valarray<T>&, const T&);
template<class T>
    requires EqualityComparable<T>
    valarray<bool> operator==(const T&, const valarray<T>&);
template<class T>
    requires EqualityComparable<T>

```

```

    valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);
template<class T>
    requires EqualityComparable<T>
    valarray<bool> operator!=(const valarray<T>&, const T&);
template<class T>
    requires EqualityComparable<T>
    valarray<bool> operator!=(const T&, const valarray<T>&);

template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator< (const valarray<T>&, const T&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator< (const T&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator> (const valarray<T>&, const T&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator> (const T&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator<=(const valarray<T>&, const T&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator<=(const T&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator>=(const valarray<T>&, const T&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator>=(const T&, const valarray<T>&);

template<classHasAbs T> valarray<T> abs (const valarray<T>&);
template<classHasAcos T> valarray<T> acos (const valarray<T>&);
template<classHasAsin T> valarray<T> asin (const valarray<T>&);
template<classHasAtan T> valarray<T> atan (const valarray<T>&);

```

```

template<classHasAtan2 T> valarray<T> atan2
    (const valarray<T>&, const valarray<T>&);
template<classHasAtan2 T> valarray<T> atan2(const valarray<T>&, const T&);
template<classHasAtan2 T> valarray<T> atan2(const T&, const valarray<T>&);

template<classHasCos T> valarray<T> cos (const valarray<T>&);
template<classHasCosh T> valarray<T> cosh (const valarray<T>&);
template<classHasExp T> valarray<T> exp (const valarray<T>&);
template<classHasLog T> valarray<T> log (const valarray<T>&);
template<classHasLog10 T> valarray<T> log10(const valarray<T>&);

template<classHasPow T> valarray<T> pow(const valarray<T>&, const valarray<T>&);
template<classHasPow T> valarray<T> pow(const valarray<T>&, const T&);
template<classHasPow T> valarray<T> pow(const T&, const valarray<T>&);

template<classHasSin T> valarray<T> sin (const valarray<T>&);
template<classHasSinh T> valarray<T> sinh (const valarray<T>&);
template<classHasSqrt T> valarray<T> sqrt (const valarray<T>&);
template<classHasTan T> valarray<T> tan (const valarray<T>&);
template<classHasTanh T> valarray<T> tanh (const valarray<T>&);
}

```

- 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.<sup>2)</sup>
- 4 Implementations introducing such replacement types shall provide additional functions and operators as follows:
  - for every function taking a `const valarray<T>&`, identical functions taking the replacement types shall be added;
  - 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 computed assignments of such types to `valarray<T>`, `slice_array<T>`, `gslice_array<T>`, `mask_array<T>` and `indirect_array<T>` objects.
- 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.

<sup>2)</sup> Clause ?? recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for `valarray` expressions.

## 26.5.2 Class template valarray

[template.valarray]

```

namespace std {
  template<class Semiregular T> requires DefaultConstructible<T>
  class valarray {
  public:
    typedef T value_type;

    // 26.5.2.1 construct/destroy:
    valarray();
    explicit valarray(size_t);
    valarray(const T&, size_t);
    valarray(const T*, size_t);
    valarray(const valarray&);
    valarray(valarray&&);
    valarray(const slice_array<T>&);
    valarray(const gslice_array<T>&);
    valarray(const mask_array<T>&);
    valarray(const indirect_array<T>&);
    ~valarray();

    // 26.5.2.2 assignment:
    valarray<T>& operator=(const valarray<T>&);
    valarray<T>& operator=(valarray<T>&&);
    valarray<T>& operator=(const T&);
    valarray<T>& operator=(const slice_array<T>&);
    valarray<T>& operator=(const gslice_array<T>&);
    valarray<T>& operator=(const mask_array<T>&);
    valarray<T>& operator=(const indirect_array<T>&);

    // 26.5.2.3 element access:
    const T& operator[] (size_t) const;
    T& operator[] (size_t);

    // 26.5.2.4 subset operations:
    valarray<T> operator[] (slice) const;
    slice_array<T> operator[] (slice);
    valarray<T> operator[] (const gslice&) const;
    gslice_array<T> operator[] (const gslice&);
    valarray<T> operator[] (const valarray<bool>&) const;
    mask_array<T> operator[] (const valarray<bool>&);
    valarray<T> operator[] (const valarray<size_t>&) const;
    indirect_array<T> operator[] (const valarray<size_t>&);

    // 26.5.2.5 unary operators:
    requires HasUnaryPlus<T> && Convertible<HasUnaryPlus<T>::result_type, T>
    valarray<T> operator+() const;
    requires HasNegate<T> && Convertible<HasNegate<T>::result_type, T>
    valarray<T> operator-() const;
    requires HasComplement<T> && Convertible<HasComplement<T>::result_type, T>

```

```

    valarray<T> operator~() const;
    requires HasLogicalNot<T> valarray<bool> operator!() const;

// 26.5.2.6 computed assignment:
requires HasMultiplyAssign<T, const T&>    valarray<T>& operator*= (const T&);
requires HasDivideAssign<T, const T&>      valarray<T>& operator/= (const T&);
requires HasModulusAssign<T, const T&>     valarray<T>& operator%= (const T&);
requires HasPlusAssign<T, const T&>       valarray<T>& operator+= (const T&);
requires HasMinusAssign<T, const T&>      valarray<T>& operator-= (const T&);
requires HasBitXorAssign<T, const T&>     valarray<T>& operator^= (const T&);
requires HasBitAndAssign<T, const T&>     valarray<T>& operator&= (const T&);
requires HasBitOrAssign<T, const T&>      valarray<T>& operator|= (const T&);
requires HasLeftShiftAssign<T, const T&>  valarray<T>& operator<<=(const T&);
requires HasRightShiftAssign<T, const T&> valarray<T>& operator>>=(const T&);

requires HasMultiplyAssign<T, const T&>    valarray<T>& operator*= (const valarray<T>&);
requires HasDivideAssign<T, const T&>      valarray<T>& operator/= (const valarray<T>&);
requires HasModulusAssign<T, const T&>     valarray<T>& operator%= (const valarray<T>&);
requires HasPlusAssign<T, const T&>       valarray<T>& operator+= (const valarray<T>&);
requires HasMinusAssign<T, const T&>      valarray<T>& operator-= (const valarray<T>&);
requires HasBitXorAssign<T, const T&>     valarray<T>& operator^= (const valarray<T>&);
requires HasBitOrAssign<T, const T&>     valarray<T>& operator|= (const valarray<T>&);
requires HasBitAndAssign<T, const T&>     valarray<T>& operator&= (const valarray<T>&);
requires HasLeftShiftAssign<T, const T&>  valarray<T>& operator<<=(const valarray<T>&);
requires HasRightShiftAssign<T, const T&> valarray<T>& operator>>=(const valarray<T>&);

// 26.5.2.7 member functions:
void swap(valarray&&);

size_t size() const;

requires HasPlusAssign<T, const T&>      T    sum() const;
requires LessThanComparable<T> T    min() const;
requires LessThanComparable<T> T    max() const;

valarray<T> shift (int) const;
valarray<T> cshift(int) const;
valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;
void resize(size_t sz, T c = 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. 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.<sup>3)</sup>

<sup>3)</sup> The intent is to specify an array template that has the minimum functionality necessary to address aliasing ambiguities and the proliferation of temporaries. Thus, the `valarray` template is neither a matrix class nor a field class. However, it is a very useful building block for designing such

- 2 An implementation is permitted to qualify any of the functions declared in `<valarray>` as `inline`.

### 26.5.2.1 `valarray` constructors

[`valarray.cons`]

```
valarray();
```

- 1 *Effects:* Constructs an object of class `valarray<T>`,<sup>4)</sup> which has zero length until it is passed into a library function as a modifiable lvalue or through a non-constant `this` pointer.<sup>5)</sup>

```
explicit valarray(size_t);
```

- 2 The array created by this constructor has a length equal to the value of the argument. The elements of the array are constructed using the default constructor for the instantiating type  $T$ .

```
valarray(const T&, size_t);
```

- 3 The array created by this constructor has a length equal to the second argument. The elements of the array are initialized with the value of the first argument.

```
valarray(const T*, size_t);
```

- 4 The array created by this constructor has a length equal to the second argument  $n$ . The values of the elements of the array are initialized with the first  $n$  values pointed to by the first argument.<sup>6)</sup> If the value of the second argument is greater than the number of values pointed to by the first argument, the behavior is undefined.

```
valarray(const valarray<T>&);
```

- 5 The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array.<sup>7)</sup>

```
valarray(valarray<T>&&);
```

- 6 The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array. After construction, `v` is in a valid but unspecified state.

- 7 *Complexity:* Constant.

- 8 *Throws:* Nothing.

```
valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);
```

- 9 These conversion constructors convert one of the four reference templates to a `valarray`.

---

classes.

<sup>4)</sup> For convenience, such objects are referred to as “arrays” throughout the remainder of 26.5.

<sup>5)</sup> This default constructor is essential, since arrays of `valarray` are likely to prove useful. There shall also be a way to change the size of an array after initialization; this is supplied by the semantics of the `resize` member function.

<sup>6)</sup> This constructor is the preferred method for converting a C array to a `valarray` object.

<sup>7)</sup> This copy constructor creates a distinct array rather than an alias. Implementations in which arrays share storage are permitted, but they shall implement a copy-on-reference mechanism to ensure that arrays are conceptually distinct.

```
~valarray();
```

10 The destructor is applied to every element of `*this`; an implementation may return all allocated memory.

### 26.5.2.2 `valarray` assignment

[`valarray.assign`]

```
valarray<T>& operator=(const valarray<T>&);
```

1 Each element of the `*this` array is assigned the value of the corresponding element of the argument array. The resulting behavior is undefined if the length of the argument array is not equal to the length of the `*this` array.

```
valarray<T>& operator=(valarray<T>&&);
```

2 *Effects:* `*this` obtains the value of `v`. After the assignment, `v` is in a valid but unspecified state.

3 *Complexity:* Constant.

4 *Throws:* Nothing.

```
valarray<T>& operator=(const T&);
```

5 The scalar assignment operator causes each element of the `*this` array to be assigned the value of the argument.

```
valarray<T>& operator=(const slice_array<T>&);
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);
```

6 These operators allow the results of a generalized subscripting operation to be assigned directly to a `valarray`.

7 If the value of an element in the left-hand side of a `valarray` assignment operator depends on the value of another element in that left-hand side, the resulting behavior is undefined.

### 26.5.2.3 `valarray` element access

[`valarray.access`]

```
const T& operator[](size_t) const;
T& operator[](size_t);
```

1 When applied to a constant array, the subscript operator returns the value of the corresponding element of the array. When applied to a non-constant array, the subscript operator returns a reference to the corresponding element of the array.

2 Thus, the expression `(a[i] = q, a[i]) == q` evaluates as 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`.

3 The expression `&a[i+j] == &a[i] + j` evaluates as true for all `size_t` `i` and `size_t` `j` such that `i+j` is less than the length of the non-constant array `a`.

4 Likewise, the expression `&a[i] != &b[j]` evaluates as true for any two non-constant arrays `a` and `b` and for any `size_t` `i` and `size_t` `j` such that `i` is less than the length of `a` and `j` is less than the length of `b`. This property

indicates an absence of aliasing and may be used to advantage by optimizing compilers.<sup>8)</sup>

- 5 The reference returned by the subscript operator for a non-constant array is guaranteed to be valid until the member function `resize(size_t, T)` (26.5.2.7) is called for that array or until the lifetime of that array ends, whichever happens first.
- 6 If the subscript operator is invoked with a `size_t` argument whose value is not less than the length of the array, the behavior is undefined.

#### 26.5.2.4 `valarray` subset operations

[`valarray.sub`]

```

valarray<T>      operator[] (slice) const;
slice_array<T>  operator[] (slice);
valarray<T>      operator[] (const gslice&) const;
gslice_array<T> operator[] (const gslice&);
valarray<T>      operator[] (const valarray<bool>&) const;
mask_array<T>   operator[] (const valarray<bool>&);
valarray<T>      operator[] (const valarray<size_t>&) const;
indirect_array<T> operator[] (const valarray<size_t>&);

```

- 1 Each of these operations returns a subset of the array. The `const`-qualified versions return this subset as a new `valarray`. The non-`const` versions return a class template object which has reference semantics to the original array.

#### 26.5.2.5 `valarray` unary operators

[`valarray.unary`]

```

requires HasUnaryPlus<T> && Convertible<HasUnaryPlus<T>::result_type, T>
valarray<T> operator+() const;
requires HasNegate<T> && Convertible<HasNegate<T>::result_type, T>
valarray<T> operator-() const;
requires HasComplement<T> && Convertible<HasComplement<T>::result_type, T>
valarray<T> operator~() const;
requires HasLogicalNot<T> valarray<bool> operator!() const;

```

- 1 ~~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 converted to type  $T$  (*bool* for *operator!*).~~
- 2 Each of these operators returns an array whose length is equal to the length of the array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

#### 26.5.2.6 `valarray` computed assignment

[`valarray.cassign`]

<sup>8)</sup> Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from `operator new`, and other techniques to generate efficient `valarrays`.

```

requires HasMultiplyAssign<T, const T&> valarray<T>& operator*=(const valarray<T>&);
requires HasDivideAssign<T, const T&> valarray<T>& operator/=(const valarray<T>&);
requires HasModulusAssign<T, const T&> valarray<T>& operator%=(const valarray<T>&);
requires HasPlusAssign<T, const T&> valarray<T>& operator+=(const valarray<T>&);
requires HasMinusAssign<T, const T&> valarray<T>& operator-=(const valarray<T>&);
requires HasBitXorAssign<T, const T&> valarray<T>& operator^=(const valarray<T>&);
requires HasBitAndAssign<T, const T&> valarray<T>& operator&=(const valarray<T>&);
requires HasBitOrAssign<T, const T&> valarray<T>& operator|=(const valarray<T>&);
requires HasLeftShiftAssign<T, const T&> valarray<T>& operator<<=(const valarray<T>&);
requires HasRightShiftAssign<T, const T&> valarray<T>& operator>>=(const valarray<T>&);

```

1 ~~Each of these operators may only be instantiated for a type  $T$  to which the indicated operator can be applied. Each of these operators performs the indicated operation on each of its elements and the corresponding element of the argument array.~~

2 The array is then returned by reference.

3 If the array and the argument array do not have the same length, the behavior is undefined. The appearance of an array on the left-hand side of a computed assignment does *not* invalidate references or pointers.

4 If the value of an element in the left-hand side of a valarray computed assignment operator depends on the value of another element in that left hand side, the resulting behavior is undefined.

```

requires HasMultiplyAssign<T, const T&> valarray<T>& operator*=(const T&);
requires HasDivideAssign<T, const T&> valarray<T>& operator/=(const T&);
requires HasModulusAssign<T, const T&> valarray<T>& operator%=(const T&);
requires HasPlusAssign<T, const T&> valarray<T>& operator+=(const T&);
requires HasMinusAssign<T, const T&> valarray<T>& operator-=(const T&);
requires HasBitXorAssign<T, const T&> valarray<T>& operator^=(const T&);
requires HasBitAndAssign<T, const T&> valarray<T>& operator&=(const T&);
requires HasBitOrAssign<T, const T&> valarray<T>& operator|=(const T&);
requires HasLeftShiftAssign<T, const T&> valarray<T>& operator<<=(const T&);
requires HasRightShiftAssign<T, const T&> valarray<T>& operator>>=(const T&);

```

5 ~~Each of these operators may only be instantiated for a type  $T$  to which the indicated operator can be applied.~~

6 Each of these operators applies the indicated operation to each element of the array and the non-array argument.

7 The array is then returned by reference.

8 The appearance of an array on the left-hand side of a computed assignment does *not* invalidate references or pointers to the elements of the array.

### 26.5.2.7 valarray member functions

[valarray.members]

```
void swap(valarray&& v);
```

1 *Effects:* \*this obtains the value of v. v obtains the value of \*this.

2 *Complexity:* Constant.

3 *Throws:* Nothing.

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

- 4 This function returns the number of elements in the array.

```
requires HasPlusAssign<T, const T&> T sum() const;
```

~~This function may only be instantiated for a type  $T$  to which operator $+$  can be applied.~~ This function returns the sum of all the elements of the array.

- 5 If the array has length 0, the behavior is undefined. If the array has length 1, `sum()` 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.

```
requires LessThanComparable<T> T min() const;
```

- 6 This function returns the minimum value contained in `*this`. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator $<$ .

```
requires LessThanComparable<T> T max() const;
```

- 7 This function returns the maximum value contained in `*this`. The value returned for an array of length 0 is undefined. 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<T> shift(int n) const;
```

- 8 This function returns an object of class `valarray<T>` of length `size()`, each of whose elements  $I$  is `(*this)[I + n]` if  $I + n$  is non-negative and less than `size()`, otherwise `T()`. Thus if element zero is taken as the leftmost element, a positive value of  $n$  shifts the elements left  $n$  places, with zero fill.

- 9 [*Example:* If the argument has the value -2, the first two elements of the result will be constructed using the default constructor; the third element of the result will be assigned the value of the first element of the argument; etc. — *end example* ]

```
valarray<T> cshift(int n) const;
```

- 10 This function returns an object of class `valarray<T>`, of length `size()`, each of whose elements  $I$  is `(*this)[(I + n) % size()]`. Thus, if element zero is taken as the leftmost element, a positive value of  $n$  shifts the elements circularly left  $n$  places.

```
valarray<T> apply(T func(T)) const;
```

```
valarray<T> apply(T func(const T&)) const;
```

- 11 These functions return an array whose length is equal to the array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the array.

```
void resize(size_t sz, T c = T());
```

- 12 This member function 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.

## 26.5.3 valarray non-member operations

[valarray.nonmembers]

## 26.5.3.1 valarray binary operators

[valarray.binary]

```

template<class T>
    requires HasMultiply<T, T> && Convertible<HasMultiply<T, T>::result_type, T>
    valarray<T> operator*(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasDivide<T, T> && Convertible<HasDivide<T, T>::result_type, T>
    valarray<T> operator/(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasModulus<T, T> && Convertible<HasModulus<T, T>::result_type, T>
    valarray<T> operator%(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasPlus<T, T> && Convertible<HasPlus<T, T>::result_type, T>
    valarray<T> operator+(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasMinus<T, T> && Convertible<HasMinus<T, T>::result_type, T>
    valarray<T> operator-(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasBitXor<T, T> && Convertible<HasBitXor<T, T>::result_type, T>
    valarray<T> operator^(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasBitAnd<T, T> && Convertible<HasBitAnd<T, T>::result_type, T>
    valarray<T> operator&(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasBitOr<T, T> && Convertible<HasBitOr<T, T>::result_type, T>
    valarray<T> operator|(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasLeftShift<T, T> && Convertible<HasLeftShift<T, T>::result_type, T>
    valarray<T> operator<<(const valarray<T>&, const valarray<T>&);
template<class T>
    requires HasRightShift<T, T> && Convertible<HasRightShift<T, T>::result_type, T>
    valarray<T> operator>>(const valarray<T>&, const valarray<T>&);

```

- 1 **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 converted to type  $T$ .**
- 2 Each of these operators returns an array 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.
- 3 If the argument arrays do not have the same length, the behavior is undefined.

```

template<class T>
    requires HasMultiply<T, T> && Convertible<HasMultiply<T, T>::result_type, T>
    valarray<T> operator*(const valarray<T>&, const T&);
template<class T>
    requires HasMultiply<T, T> && Convertible<HasMultiply<T, T>::result_type, T>
    valarray<T> operator*(const T&, const valarray<T>&);

```

```

template<class T>
    requires HasDivide<T, T> && Convertible<HasDivide<T, T>::result_type, T>
    valarray<T> operator/ (const valarray<T>&, const T&);
template<class T>
    requires HasDivide<T, T> && Convertible<HasDivide<T, T>::result_type, T>
    valarray<T> operator/ (const T&, const valarray<T>&);
template<class T>
    requires HasModulus<T, T> && Convertible<HasModulus<T, T>::result_type, T>
    valarray<T> operator% (const valarray<T>&, const T&);
template<class T>
    requires HasModulus<T, T> && Convertible<HasModulus<T, T>::result_type, T>
    valarray<T> operator% (const T&, const valarray<T>&);
template<class T>
    requires HasPlus<T, T> && Convertible<HasPlus<T, T>::result_type, T>
    valarray<T> operator+ (const valarray<T>&, const T&);
template<class T>
    requires HasPlus<T, T> && Convertible<HasPlus<T, T>::result_type, T>
    valarray<T> operator+ (const T&, const valarray<T>&);
template<class T>
    requires HasMinus<T, T> && Convertible<HasMinus<T, T>::result_type, T>
    valarray<T> operator- (const valarray<T>&, const T&);
template<class T>
    requires HasMinus<T, T> && Convertible<HasMinus<T, T>::result_type, T>
    valarray<T> operator- (const T&, const valarray<T>&);
template<class T>
    requires HasBitXor<T, T> && Convertible<HasBitXor<T, T>::result_type, T>
    valarray<T> operator^ (const valarray<T>&, const T&);
template<class T>
    requires HasBitXor<T, T> && Convertible<HasBitXor<T, T>::result_type, T>
    valarray<T> operator^ (const T&, const valarray<T>&);
template<class T>
    requires HasBitAnd<T, T> && Convertible<HasBitAnd<T, T>::result_type, T>
    valarray<T> operator& (const valarray<T>&, const T&);
template<class T>
    requires HasBitAnd<T, T> && Convertible<HasBitAnd<T, T>::result_type, T>
    valarray<T> operator& (const T&, const valarray<T>&);
template<class T>
    requires HasBitOr<T, T> && Convertible<HasBitOr<T, T>::result_type, T>
    valarray<T> operator| (const valarray<T>&, const T&);
template<class T>
    requires HasBitOr<T, T> && Convertible<HasBitOr<T, T>::result_type, T>
    valarray<T> operator| (const T&, const valarray<T>&);
template<class T>
    requires HasLeftShift<T, T> && Convertible<HasLeftShift<T, T>::result_type, T>
    valarray<T> operator<<(const valarray<T>&, const T&);
template<class T>
    requires HasLeftShift<T, T> && Convertible<HasLeftShift<T, T>::result_type, T>
    valarray<T> operator<<(const T&, const valarray<T>&);
template<class T>
    requires HasRightShift<T, T> && Convertible<HasRightShift<T, T>::result_type, T>

```

```

valarray<T> operator>>(const valarray<T>&, const T&);
template<class T>
  requires HasRightShift<T, T> && Convertible<HasRightShift<T, T>::result_type, T>
  valarray<T> operator>>(const T&, const valarray<T>&);

```

4 ~~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 converted to type  $T$ .~~

5 Each of these operators returns an array 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.

### 26.5.3.2 valarray logical operators

[valarray.comparison]

```

template<class T>
  requires EqualityComparable<T>
  valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
template<class T>
  requires EqualityComparable<T>
  valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);
template<class T>
  requires LessThanComparable<T>
  valarray<bool> operator<(const valarray<T>&, const valarray<T>&);
template<class T>
  requires LessThanComparable<T>
  valarray<bool> operator>(const valarray<T>&, const valarray<T>&);
template<class T>
  requires LessThanComparable<T>
  valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
template<class T>
  requires LessThanComparable<T>
  valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
template<class T>
  requires HasLogicalAnd<T, T>
  valarray<bool> operator&&(const valarray<T>&, const valarray<T>&);
template<class T>
  requires HasLogicalOr<T, T>
  valarray<bool> operator|| (const valarray<T>&, const valarray<T>&);

```

1 ~~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 converted to type `bool`.~~

2 Each of these operators returns a `bool` array 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.

3 If the two array arguments do not have the same length, the behavior is undefined.

```

template<class T>
    requires EqualityComparable<T>
    valarray<bool> operator==(const valarray<T>&, const T&);
template<class T>
    requires EqualityComparable<T>
    valarray<bool> operator==(const T&, const valarray<T>&);
template<class T>
    requires EqualityComparable<T>
    valarray<bool> operator!=(const valarray<T>&, const T&);
template<class T>
    requires EqualityComparable<T>
    valarray<bool> operator!=(const T&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator< (const valarray<T>&, const T&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator< (const T&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator> (const valarray<T>&, const T&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator> (const T&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator<= (const valarray<T>&, const T&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator<= (const T&, const valarray<T>&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator>= (const valarray<T>&, const T&);
template<class T>
    requires LessThanComparable<T>
    valarray<bool> operator>= (const T&, const valarray<T>&);
template<class T>
    requires HasLogicalAnd<T, T>
    valarray<bool> operator&&(const valarray<T>&, const T&);
template<class T>
    requires HasLogicalAnd<T, T>
    valarray<bool> operator&&(const T&, const valarray<T>&);
template<class T>
    requires HasLogicalOr<T, T>
    valarray<bool> operator||(const valarray<T>&, const T&);
template<class T>
    requires HasLogicalOr<T, T>
    valarray<bool> operator||(const T&, const valarray<T>&);

```

4 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 converted to type `bool`.

- 5 Each of these operators returns a `bool` array 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.

### 26.5.3.3 `valarray` transcendentals

[`valarray.transcend`]

```
template<class HasAbs T> valarray<T> abs (const valarray<T>&);
template<class HasAcos T> valarray<T> acos (const valarray<T>&);
template<class HasAsin T> valarray<T> asin (const valarray<T>&);
template<class HasAtan T> valarray<T> atan (const valarray<T>&);
template<class HasAtan2 T> valarray<T> atan2
    (const valarray<T>&, const valarray<T>&);
template<class HasAtan2 T> valarray<T> atan2(const valarray<T>&, const T&);
template<class HasAtan2 T> valarray<T> atan2(const T&, const valarray<T>&);
template<class HasCos T> valarray<T> cos (const valarray<T>&);
template<class HasCosh T> valarray<T> cosh (const valarray<T>&);
template<class HasExp T> valarray<T> exp (const valarray<T>&);
template<class HasLog T> valarray<T> log (const valarray<T>&);
template<class HasLog10 T> valarray<T> log10(const valarray<T>&);
template<class HasPow T> valarray<T> pow
    (const valarray<T>&, const valarray<T>&);
template<class HasPow T> valarray<T> pow (const valarray<T>&, const T&);
template<class HasPow T> valarray<T> pow (const T&, const valarray<T>&);
template<class HasSin T> valarray<T> sin (const valarray<T>&);
template<class HasSinh T> valarray<T> sinh (const valarray<T>&);
template<class HasSqrt T> valarray<T> sqrt (const valarray<T>&);
template<class HasTan T> valarray<T> tan (const valarray<T>&);
template<class HasTanh T> valarray<T> tanh (const valarray<T>&);
```

- 1 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 which is of type  $T$  or which can be unambiguously converted to type  $T$ . Each element of the returned array is initialized with the result of applying the indicated function to the non-array argument (if any) and corresponding element of the array argument or arguments.

### 26.5.3.4 `valarray` specialized algorithms

[`valarray.special`]

```
template <class Semiregular T> void swap(valarray<T>& x, valarray<T>& y);
template <class Semiregular T> void swap(valarray<T>&& x, valarray<T>& y);
template <class Semiregular T> void swap(valarray<T>& x, valarray<T>&& y);
```

- 1 *Effects:* `x.swap(y)`.

### 26.5.4 Class slice

[`class.slice`]

### 26.5.5 Class template `slice_array`

[`template.slice.array`]

```

namespace std {
    template <class Semiregular T> class slice_array {
    public:
        typedef T value_type;

        void operator= (const valarray<T>&) const;
        requires HasMultiplyAssign<T, const T&> void operator*= (const valarray<T>&) const;
        requires HasDivideAssign<T, const T&> void operator/= (const valarray<T>&) const;
        requires HasModulusAssign<T, const T&> void operator%=(const valarray<T>&) const;
        requires HasPlusAssign<T, const T&> void operator+=(const valarray<T>&) const;
        requires HasMinusAssign<T, const T&> void operator-=(const valarray<T>&) const;
        requires HasBitXorAssign<T, const T&> void operator^=(const valarray<T>&) const;
        requires HasBitAndAssign<T, const T&> void operator&=(const valarray<T>&) const;
        requires HasBitOrAssign<T, const T&> void operator|=(const valarray<T>&) const;
        requires HasLeftShiftAssign<T, const T&> void operator<<=(const valarray<T>&) const;
        requires HasRightShiftAssign<T, const T&> void operator>>=(const valarray<T>&) const;

        slice_array(const slice_array&);
        ~slice_array();
        slice_array& operator=(const slice_array&);
        void operator=(const T&) const;
    private:
        slice_array();
    };
}

```

- 1 The `slice_array` template is a helper template used by the `slice` subscript operator

```
slice_array<T> valarray<T>::operator[](slice);
```

It has reference semantics to a subset of an array specified by a `slice` object.

- 2 [*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, ..., 13. — *end example* ]

### 26.5.5.1 `slice_array` constructors

[**cons.slice.arr**]

```
slice_array();
```

- 1 This constructor is declared to be private. This constructor need not be defined.

### 26.5.5.2 `slice_array` assignment

[**slice.arr.assign**]

```
void operator=(const valarray<T>&) const;
slice_array& operator=(const slice_array&);
```

- 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.

**26.5.5.3 slice\_array computed assignment****[slice.arr.comp.assign]**

```

requires HasMultiplyAssign<T, const T&> void operator*=(const valarray<T>&) const;
requires HasDivideAssign<T, const T&> void operator/=(const valarray<T>&) const;
requires HasModulusAssign<T, const T&> void operator%=(const valarray<T>&) const;
requires HasPlusAssign<T, const T&> void operator+=(const valarray<T>&) const;
requires HasMinusAssign<T, const T&> void operator-=(const valarray<T>&) const;
requires HasBitXorAssign<T, const T&> void operator^=(const valarray<T>&) const;
requires HasBitAndAssign<T, const T&> void operator&=(const valarray<T>&) const;
requires HasBitOrAssign<T, const T&> void operator|=(const valarray<T>&) const;
requires HasLeftShiftAssign<T, const T&> void operator<<=(const valarray<T>&) const;
requires HasRightShiftAssign<T, const T&> void operator>>=(const valarray<T>&) const;

```

- 1 These computed 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.

**26.5.5.4 slice\_array 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.

**26.5.6 The gslice class****[class.gslice]****26.5.7 Class template gslice\_array****[template.gslice.array]**

```

namespace std {
template <class Semiregular T> class gslice_array {
public:
    typedef T value_type;

    void operator= (const valarray<T>&) const;
    requires HasMultiplyAssign<T, const T&> void operator*=(const valarray<T>&) const;
    requires HasDivideAssign<T, const T&> void operator/=(const valarray<T>&) const;
    requires HasModulusAssign<T, const T&> void operator%=(const valarray<T>&) const;
    requires HasPlusAssign<T, const T&> void operator+=(const valarray<T>&) const;
    requires HasMinusAssign<T, const T&> void operator-=(const valarray<T>&) const;
    requires HasBitXorAssign<T, const T&> void operator^=(const valarray<T>&) const;
    requires HasBitAndAssign<T, const T&> void operator&=(const valarray<T>&) const;
    requires HasBitOrAssign<T, const T&> void operator|=(const valarray<T>&) const;
    requires HasLeftShiftAssign<T, const T&> void operator<<=(const valarray<T>&) const;
    requires HasRightShiftAssign<T, const T&> void operator>>=(const valarray<T>&) const;

    gslice_array(const gslice_array&);
    ~gslice_array();
    gslice_array& operator=(const gslice_array&);
    void operator=(const T&) const;
private:

```

```

    gslice_array();
};
}

```

1 This template is a helper template used by the `slice` 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.

3 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`.

### 26.5.7.1 `gslice_array` constructors

[[gslice.array.cons](#)]

```
gslice_array();
```

1 This constructor is declared to be private. This constructor need not be defined.

### 26.5.7.2 `gslice_array` assignment

[[gslice.array.assign](#)]

```
void operator=(const valarray<T>&) const;
gslice_array& operator=(const gslice_array&);
```

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.

### 26.5.7.3 `gslice_array`

[[gslice.array.comp.assign](#)]

```

requires HasMultiplyAssign<T, const T&> void operator*=(const valarray<T>&) const;
requires HasDivideAssign<T, const T&> void operator/=(const valarray<T>&) const;
requires HasModulusAssign<T, const T&> void operator%=(const valarray<T>&) const;
requires HasPlusAssign<T, const T&> void operator+=(const valarray<T>&) const;
requires HasMinusAssign<T, const T&> void operator-=(const valarray<T>&) const;
requires HasBitXorAssign<T, const T&> void operator^=(const valarray<T>&) const;
requires HasBitAndAssign<T, const T&> void operator&=(const valarray<T>&) const;
requires HasBitOrAssign<T, const T&> void operator|=(const valarray<T>&) const;
requires HasLeftShiftAssign<T, const T&> void operator<<=(const valarray<T>&) const;
requires HasRightShiftAssign<T, const T&> void operator>>=(const valarray<T>&) const;

```

1 These computed 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.

### 26.5.7.4 `gslice_array` 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.

### 26.5.8 Class template `mask_array`

[template.mask.array]

```

namespace std {
  template <class Semiregular T> class mask_array {
  public:
    typedef T value_type;

    void operator= (const valarray<T>&) const;
    requires HasMultiplyAssign<T, const T&> void operator*= (const valarray<T>&) const;
    requires HasDivideAssign<T, const T&> void operator/= (const valarray<T>&) const;
    requires HasModulusAssign<T, const T&> void operator%= (const valarray<T>&) const;
    requires HasPlusAssign<T, const T&> void operator+= (const valarray<T>&) const;
    requires HasMinusAssign<T, const T&> void operator-= (const valarray<T>&) const;
    requires HasBitXorAssign<T, const T&> void operator^= (const valarray<T>&) const;
    requires HasBitAndAssign<T, const T&> void operator&= (const valarray<T>&) const;
    requires HasBitOrAssign<T, const T&> void operator|= (const valarray<T>&) const;
    requires HasLeftShiftAssign<T, const T&> void operator<<= (const valarray<T>&) const;
    requires HasRightShiftAssign<T, const T&> void operator>>= (const valarray<T>&) const;

    mask_array(const mask_array&);
    ~mask_array();
    mask_array& operator=(const mask_array&);
    void operator=(const T&) const;
  private:
    mask_array();
  };
}

```

- 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.)

#### 26.5.8.1 `mask_array` constructors

[mask.array.cons]

```
mask_array();
```

- 1 This constructor is declared to be private. This constructor need not be defined.

#### 26.5.8.2 `mask_array` assignment

[mask.array.assign]

```
void operator=(const valarray<T>&) const;
mask_array& operator=(const mask_array&);
```

- 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.

### 26.5.8.3 `mask_array` computed assignment

[`mask.array.comp.assign`]

```
requires HasMultiplyAssign<T, const T&> void operator*= (const valarray<T>&) const;
requires HasDivideAssign<T, const T&> void operator/= (const valarray<T>&) const;
requires HasModulusAssign<T, const T&> void operator%=(const valarray<T>&) const;
requires HasPlusAssign<T, const T&> void operator+=(const valarray<T>&) const;
requires HasMinusAssign<T, const T&> void operator-=(const valarray<T>&) const;
requires HasBitXorAssign<T, const T&> void operator^=(const valarray<T>&) const;
requires HasBitAndAssign<T, const T&> void operator&=(const valarray<T>&) const;
requires HasBitOrAssign<T, const T&> void operator|=(const valarray<T>&) const;
requires HasLeftShiftAssign<T, const T&> void operator<<=(const valarray<T>&) const;
requires HasRightShiftAssign<T, const T&> void operator>>=(const valarray<T>&) const;
```

- 1 These computed 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.

### 26.5.8.4 `mask_array` 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.

### 26.5.9 Class template `indirect_array`

[`template.indirect.array`]

```
namespace std {
  template <class Semiregular T> class indirect_array {
  public:
    typedef T value_type;

    void operator= (const valarray<T>&) const;
    requires HasMultiplyAssign<T, const T&> void operator*= (const valarray<T>&) const;
    requires HasDivideAssign<T, const T&> void operator/= (const valarray<T>&) const;
    requires HasModulusAssign<T, const T&> void operator%=(const valarray<T>&) const;
    requires HasPlusAssign<T, const T&> void operator+=(const valarray<T>&) const;
    requires HasMinusAssign<T, const T&> void operator-=(const valarray<T>&) const;
    requires HasBitXorAssign<T, const T&> void operator^=(const valarray<T>&) const;
    requires HasBitAndAssign<T, const T&> void operator&=(const valarray<T>&) const;
    requires HasBitOrAssign<T, const T&> void operator|=(const valarray<T>&) const;
    requires HasLeftShiftAssign<T, const T&> void operator<<=(const valarray<T>&) const;
    requires HasRightShiftAssign<T, const T&> void operator>>=(const valarray<T>&) const;
```

```

    indirect_array(const indirect_array&);
    ~indirect_array();
    indirect_array& operator=(const indirect_array&);
    void operator=(const T&) const;
private:
    indirect_array();
};
}

```

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`.

### 26.5.9.1 `indirect_array` constructors

[[indirect.array.cons](#)]

```
indirect_array();
```

1 This constructor is declared to be private. This constructor need not be defined.

### 26.5.9.2 `indirect_array` assignment

[[indirect.array.assign](#)]

```
void operator=(const valarray<T>&) const;
indirect_array& operator=(const indirect_array&);
```

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* ]

### 26.5.9.3 `indirect_array` computed assignment

[[indirect.array.comp.assign](#)]

```

requires HasMultiplyAssign<T, const T&> void operator*=(const valarray<T>&) const;
requires HasDivideAssign<T, const T&> void operator/=(const valarray<T>&) const;
requires HasModulusAssign<T, const T&> void operator%=(const valarray<T>&) const;
requires HasPlusAssign<T, const T&> void operator+=(const valarray<T>&) const;

```

```

requires HasMinusAssign<T, const T&> void operator-= (const valarray<T>&) const;
requires HasBitXorAssign<T, const T&> void operator^= (const valarray<T>&) const;
requires HasBitAndAssign<T, const T&> void operator&= (const valarray<T>&) const;
requires HasBitOrAssign<T, const T&> void operator|= (const valarray<T>&) const;
requires HasLeftShiftAssign<T, const T&> void operator<<= (const valarray<T>&) const;
requires HasRightShiftAssign<T, const T&> void operator>>= (const valarray<T>&) const;

```

- 1 These computed 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.

#### 26.5.9.4 `indirect_array` 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.

## 26.6 Generalized numeric operations

[`numeric.ops`]

### Header `<numeric>` synopsis

```

namespace std {
    template <InputIterator Iter, MoveConstructible T>
        requires HasPlus<T, Iter::reference>
            && HasAssign<T, HasPlus<T, Iter::reference>::result_type>
        T accumulate(Iter first, Iter last, T init);
    template <InputIterator Iter, MoveConstructible T,
        Callable<auto, const T&, Iter::reference> BinaryOperation>
        requires HasAssign<T, BinaryOperation::result_type>
            && CopyConstructible<BinaryOperation>
        T accumulate(Iter first, Iter last, T init,
            BinaryOperation binary_op);
    template <InputIterator Iter1, InputIterator Iter2, MoveConstructible T>
        requires HasMultiply<Iter1::reference, Iter2::reference>
            && HasPlus<T, HasMultiply<Iter1::reference, Iter2::reference>::result_type>
            && HasAssign<
                T,
                HasPlus<T,
                    HasMultiply<Iter1::reference, Iter2::reference>::result_type>::result_type>
        T inner_product(Iter1 first1, Iter1 last1,
            Iter2 first2, T init);
    template <InputIterator Iter1, InputIterator Iter2, MoveConstructible T,
        class BinaryOperation1,
        Callable<auto, Iter1::reference, Iter2::reference> BinaryOperation2>
        requires Callable<BinaryOperation1, const T&, BinaryOperation2::result_type>
            && HasAssign<T, BinaryOperation1::result_type>
            && CopyConstructible<BinaryOperation1>

```

Draft

```

    && CopyConstructible<BinaryOperation2>
    T inner_product(Iter1 first1, Iter1 last1,
                   Iter2 first2, T init,
                   BinaryOperation1 binary_op1,
                   BinaryOperation2 binary_op2);
template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter>
    requires HasPlus<InIter::value_type, InIter::reference>
           && HasAssign<InIter::value_type,
                       HasPlus<InIter::value_type, InIter::reference>::result_type>
           && Constructible<InIter::value_type, InIter::reference>
    OutIter partial_sum(InIter first, InIter last,
                       OutIter result);
template<InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter,
         Callable<auto, const InIter::value_type&, InIter::reference> BinaryOperation>
    requires HasAssign<InIter::value_type, BinaryOperation::result_type>
           && Constructible<InIter::value_type, InIter::reference>
           && CopyConstructible<BinaryOperation>
    OutIter partial_sum(InIter first, InIter last,
                       OutIter result, BinaryOperation binary_op);
template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter>
    requires HasMinus<InIter::value_type, InIter::value_type>
           && Constructible<InIter::value_type, InIter::reference>
           && OutputIterator<OutIter, HasMinus<InIter::value_type, InIter::value_type>::result_type>
           && MoveAssignable<InIter::value_type>
    OutIter adjacent_difference(InIter first, InIter last,
                               OutIter result);
template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter,
         Callable<auto, const InIter::value_type&, const InIter::value_type&> BinaryOperation>
    requires Constructible<InIter::value_type, InIter::reference>
           && OutputIterator<OutIter, BinaryOperation::result_type>
           && MoveAssignable<InIter::value_type>
           && CopyConstructible<BinaryOperation>
    OutIter adjacent_difference(InIter first, InIter last,
                               OutIter result,
                               BinaryOperation binary_op);

template <ForwardIterator Iter, HasPreincrement T>
    requires OutputIterator<Iter, const T&>
    void iota(Iter first, Iter last, T value);

```

- 1 The requirements on the types of algorithms' arguments that are described in the introduction to clause ?? also apply to the following algorithms.

### 26.6.1 Accumulate

[accumulate]

```

template <InputIterator Iter, MoveConstructible T>
    requires HasPlus<T, Iter::reference>
           && HasAssign<T, HasPlus<T, Iter::reference>::result_type>
    T accumulate(Iter first, Iter last, T init);
template <InputIterator Iter, MoveConstructible T,

```

```

    Callable<auto, const T&, Iter::reference> BinaryOperation>
requires HasAssign<T, BinaryOperation::result_type>
    && CopyConstructible<BinaryOperation>
T accumulate(Iter first, Iter last, T init,
             BinaryOperation binary_op);

```

- 1 *Effects:* Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifies it with `acc = acc + *i` or `acc = binary_op(acc, *i)` for every iterator `i` in the range `[first,last)` in order.<sup>9)</sup>
- 2 *Requires:* **T shall meet the requirements of CopyConstructible (20.1.3) and Assignable (21.3) types.** In the range `[first,last]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.<sup>10)</sup>

### 26.6.2 Inner product

[inner.product]

```

template <InputIterator Iter1, InputIterator Iter2, MoveConstructible T>
requires HasMultiply<Iter1::reference, Iter2::reference>
    && HasPlus<T, HasMultiply<Iter1::reference, Iter2::reference>::result_type>
    && HasAssign<
        T,
        HasPlus<T,
            HasMultiply<Iter1::reference, Iter2::reference>::result_type>::result_type>
T inner_product(Iter1 first1, Iter1 last1,
               Iter2 first2, T init);

```

```

template <InputIterator Iter1, InputIterator Iter2, MoveConstructible T,
         class BinaryOperation1,
         Callable<auto, Iter1::reference, Iter2::reference> BinaryOperation2>
requires Callable<BinaryOperation1, const T&, BinaryOperation2::result_type>
    && HasAssign<T, BinaryOperation1::result_type>
    && CopyConstructible<BinaryOperation1>
    && CopyConstructible<BinaryOperation2>
T inner_product(Iter1 first1, Iter1 last1,
               Iter2 first2, T init,
               BinaryOperation1 binary_op1,
               BinaryOperation2 binary_op2);

```

- 1 *Effects:* Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = acc + (*i1) * (*i2)` or `acc = binary_op1(acc, binary_op2(*i1, *i2))` for every iterator `i1` in the range `[first,last)` and iterator `i2` in the range `[first2,first2 + (last - first))` in order.
- 2 *Requires:* **T shall meet the requirements of CopyConstructible (20.1.3) and Assignable (21.3) types.** In the ranges `[first,last]` and `[first2,first2 + (last - first)]` `binary_op1` and `binary_op2` shall neither modify elements nor invalidate iterators or subranges.<sup>11)</sup>

<sup>9)</sup> `accumulate` is similar to the APL reduction operator and Common Lisp `reduce` function, but it avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value.

<sup>10)</sup> The use of fully closed ranges is intentional

<sup>11)</sup> The use of fully closed ranges is intentional

## 26.6.3 Partial sum

[partial.sum]

```

template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter>
  requires HasPlus<InIter::value_type, InIter::reference>
         && HasAssign<InIter::value_type,
             HasPlus<InIter::value_type, InIter::reference>::result_type>
         && Constructible<InIter::value_type, InIter::reference>
  OutIter partial_sum(InIter first, InIter last,
                     OutIter result);
template<InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter,
  Callable<auto, const InIter::value_type&, InIter::reference> BinaryOperation>
  requires HasAssign<InIter::value_type, BinaryOperation::result_type>
         && Constructible<InIter::value_type, InIter::reference>
         && CopyConstructible<BinaryOperation>
  OutIter partial_sum(InIter first, InIter last,
                     OutIter result, BinaryOperation binary_op);

```

1 *Effects:* Assigns to every element referred to by iterator *i* in the range  $[\text{result}, \text{result} + (\text{last} - \text{first})]$  a value correspondingly equal to

$$((\dots(*\text{first} + *(\text{first} + 1)) + \dots) + *(\text{first} + (\text{i} - \text{result})))$$

or

$$\text{binary\_op}(\text{binary\_op}(\dots, \text{binary\_op}(*\text{first}, *(\text{first} + 1)), \dots), *(\text{first} + (\text{i} - \text{result})))$$

2 *Returns:*  $\text{result} + (\text{last} - \text{first})$ .

3 *Complexity:* Exactly  $(\text{last} - \text{first}) - 1$  applications of `binary_op`.

4 *Requires:* In the ranges  $[\text{first}, \text{last}]$  and  $[\text{result}, \text{result} + (\text{last} - \text{first})]$  `binary_op` shall neither modify elements nor invalidate iterators or subranges.<sup>12)</sup>

5 *Remarks:* `result` may be equal to `first`.

## 26.6.4 Adjacent difference

[adjacent.difference]

```

template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter>
  requires HasMinus<InIter::value_type, InIter::value_type>
         && Constructible<InIter::value_type, InIter::reference>
         && OutputIterator<OutIter, HasMinus<InIter::value_type, InIter::value_type>::result_type>
         && MoveAssignable<InIter::value_type>
  OutIter adjacent_difference(InIter first, InIter last,
                             OutIter result);
template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter,
  Callable<auto, const InIter::value_type&, const InIter::value_type&> BinaryOperation>
  requires Constructible<InIter::value_type, InIter::reference>
         && OutputIterator<OutIter, BinaryOperation::result_type>

```

<sup>12)</sup>The use of fully closed ranges is intentional.

```

    && MoveAssignable<InIter::value_type>
    && CopyConstructible<BinaryOperation>
    OutIter adjacent_difference(InIter first, InIter last,
                              OutIter result,
                              BinaryOperation binary_op);

```

1 *Effects:* Assigns to every element referred to by iterator *i* in the range [*result* + 1, *result* + (*last* - *first*)) a value correspondingly equal to

```
*(first + (i - result)) - *(first + (i - result) - 1)
```

or

```
binary_op(*(first + (i - result)), *(first + (i - result) - 1)).
```

*result* gets the value of *\*first*.

2 *Requires:* In the ranges [*first*, *last*] and [*result*, *result* + (*last* - *first*)], *binary\_op* shall neither modify elements nor invalidate iterators or subranges.<sup>13)</sup>

3 *Remarks:* *result* may be equal to *first*.

4 *Returns:* *result* + (*last* - *first*).

5 *Complexity:* Exactly (*last* - *first*) - 1 applications of *binary\_op*.

### 26.6.5 Iota

[numeric.iota]

```

template <ForwardIterator Iter, HasPreincrement T>
    requires OutputIterator<Iter, const T&>
    void iota(Iter first, Iter last, T value);

```

1 ~~*Requires:* T shall meet the requirements of CopyConstructible and Assignable types, and shall be convertible to ForwardIterator's value type. The expression ++*val*, where *val* has type T, shall be well formed.~~

2 *Effects:* For each element referred to by the iterator *i* in the range [*first*, *last*), assigns *\*i* = *value* and increments *value* as if by ++*value*.

3 *Complexity:* Exactly *last* - *first* increments and assignments.

### Acknowledgments

Daniel Krüger provided conceptualized versions of `iota` in N2666

### Bibliography

- [1] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek. Proposed wording for concepts (revision 8). Technical Report N2741=08-0251, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2008.

<sup>13)</sup>The use of fully closed ranges is intentional.