22

Numerics

The purpose of computing is insight, not numbers.
– R.W. Hamming

... but for the student,
numbers are often the best road to insight.
– A. Ralston

Introduction — numeric limits — mathematical functions — ***valarray*** — vector operations — slices — ***slice_array*** — elimination of temporaries — ***gslice_array*** — ***mask_array*** — ***indirect_array*** — ***complex*** — generalized algorithms — random numbers — advice — exercises.

22.1 Introduction [num.intro]

It is rare to write any real code without doing some calculation. However, most code requires little mathematics beyond simple arithmetic. This chapter presents the facilities the standard library offers to people who go beyond that.

Neither C nor C++ were designed primarily with numeric computation in mind. However, numeric computation typically occurs in the context of other work – such as database access, networking, instrument control, graphics, simulation, financial analysis, etc. – so C++ becomes an attractive vehicle for computations that are part of a larger system. Furthermore, numeric methods have come a long way from being simple loops over vectors of floating-point numbers. Where more complex data structures are needed as part of a computation, C++'s strengths become relevant. The net effect is that C++ is increasingly used for scientific and engineering computation involving sophisticated numerics. Consequently, facilities and techniques supporting such computation have emerged. This chapter describes the parts of the standard library that support numerics and presents a few techniques for dealing with issues that arise when people express numeric

computations in C++. I make no attempt to teach numeric methods. Numeric computation is a fascinating topic in its own right. To understand it, you need a good course in numerical methods or at least a good textbook – not just a language manual and tutorial.

22.2 Numeric Limits [num.limits]

To do anything interesting with numbers, we typically need to know something about general properties of built-in numeric types that are implementation-defined rather than fixed by the rules of the language itself (§4.6). For example, what is the largest ***int***? What is the smallest ***float***? Is a ***double*** rounded or truncated when assigned to a ***float***? How many bits are there in a ***char***?

Answers to such questions are provided by the specializations of the ***numeric_limits*** template presented in ***<limits>***. For example:

```
void f(double d, int i)
{
    if (numeric_limits<char>::digits != 8) {
        // unusual bytes (number of bits not 8)
    }

    if (i<numeric_limits<short>::min() || numeric_limits<short>::max()<i) {
        // i cannot be stored in a short without loss of precision
    }

    if (0<d && d<numeric_limits<double>::epsilon()) d = 0;

    if (numeric_limits<Quad>::is_specialized) {
        // limits information available for type Quad
    }
}
```

Each specialization provides the relevant information for its argument type. Thus, the general ***numeric_limits*** template is simply a notational handle for a set of constants and inline functions:

```
template<class T> class numeric_limits {
public:
    static const bool is_specialized = false; // is information available for numeric_limits<T>?

    // uninteresting defaults
};
```

The real information is in the specializations. Each implementation of the standard library provides a specialization of ***numeric_limits*** for each fundamental type (the character types, the integer and floating-point types, and ***bool***) but not for any other plausible candidates such as ***void***, enumerations, or library types (such as ***complex<double>***).

For an integral type such as ***char***, only a few pieces of information are of interest. Here is ***numeric_limits<char>*** for an implementation in which a ***char*** has 8 bits and is signed:

```
class numeric_limits<char> {
public:
    static const bool is_specialized = true;      // yes, we have information

    static const int digits = 8;                  // number of bits (''binary digits'')

    static const bool is_signed = true;           // this implementation has char signed
    static const bool is_integer = true;          // char is an integral type

    inline static char min() throw() { return -128; }    // smallest value
    inline static char max() throw() { return 127; }     // largest value

    // lots of declarations not relevant to a char
};
```

Most members of ***numeric_limits*** are intended to describe floating-point numbers. For example, this describes one possible implementation of ***float***:

```
class numeric_limits<float> {
public:
    static const bool is_specialized = true;

    static const int radix = 2;         // base of exponent (in this case, binary)
    static const int digits = 24;       // number radix digits in mantissa
    static const int digits10 = 6;      // number of base 10 digits in mantissa

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

    inline static float min() throw() { return 1.17549435E-38F; }
    inline static float max() throw() { return 3.40282347E+38F; }

    inline static float epsilon() throw() { return 1.19209290E-07F; }
    inline static float round_error() throw() { return 0.5F; }

    inline static float infinity() throw() { return /* some value */; }
    inline static float quiet_NaN() throw() { return /* some value */; }
    inline static float signaling_NaN() throw() { return /* some value */; }
    inline static float denorm_min() throw() { return min(); }

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

    static const bool has_infinity = true;
    static const bool has_quiet_NaN = true;
    static const bool has_signaling_NaN = true;
    static const float_denorm_style has_denorm = denorm_absent;    // enum from <limits>
    static const bool has_denorm_loss = false;

    static const bool is_iec559 = true;    // conforms to IEC-559
    static const bool is_bounded = true;
    static const bool is_modulo = false;
```

```
    static const bool traps = true;
    static const bool tinyness_before = true;

    static const float_round_style round_style = round_to_nearest;    // enum from <limits>
};
```

Note that ***min()*** is the smallest *positive* normalized number and that ***epsilon*** is the smallest positive floating-point number such that ***1+epsilon-1*** is representable.

When defining a scalar type along the lines of the built-in ones, it is a good idea also to provide a suitable specialization of ***numeric_limits***. For example, if I wrote a quadruple-precision type ***Quad*** or if a vendor provided an extended-precision integer ***long long***, a user could reasonably expect ***numeric_limits<Quad>*** and ***numeric_limits<long long>*** to be supplied.

One can imagine specializations of ***numeric_limits*** describing properties of user-defined types that have little to do with floating-point numbers. In such cases, it is usually better to use the general technique for describing properties of a type than to specialize ***numeric_limits*** with properties not considered in the standard. Latin1...UL float_denom_style

Floating-point values are represented as inline functions. Integral values in ***numeric_limits***, however, must be represented in a form that allows them to be used in constant expressions. That implies that they must have in-class initializers (§10.4.6.2). If you use ***static const*** members rather than enumerators for that, remember to define the ***static***s.

22.2.1 Limit Macros [num.limit.c]

From C, C++ inherited macros that describe properties of integers. These are found in ***<climits>*** and ***<limits.h>*** and have names such as ***CHAR_BIT*** and ***INT_MAX***. Similarly, ***<cfloat>*** and ***<float.h>*** define macros describing properties of floating-point numbers. They have names such as ***DBL_MIN_EXP***, ***FLT_RADIX***, and ***LDBL_MAX***.

As ever, macros are best avoided.

22.3 Standard Mathematical Functions [num.math]

The headers ***<cmath>*** and ***<math.h>*** provide what is commonly referred to as ''the usual mathematical functions:''

```
double abs(double);                 // absolute value; not in C, same as fabs()
double fabs(double);                // absolute value

double ceil(double d);              // smallest integer not less than d
double floor(double d);             // largest integer not greater than d

double sqrt(double d);              // square root of d, d must be non-negative

double pow(double d, double e);     // d to the power of e,
                                    // error if d==0 and e<=0 or if d<0 and e isn't an integer.
double pow(double d, int i);        // d to the power of i; not in C
```

```
double cos(double);                 // cosine
double sin(double);                 // sine
double tan(double);                 // tangent

double acos(double);                // arc cosine
double asin(double);                // arc sine
double atan(double);                // arc tangent
double atan2(double x, double y);   // atan(x/y)

double sinh(double);                // hyperbolic sine
double cosh(double);                // hyperbolic cosine
double tanh(double);                // hyperbolic tangent

double exp(double);                 // exponential, base e
double log(double d);               // natural (base e) logarithm, d must be > 0
double log10(double d);             // base 10 logarithm, d must be > 0

double modf(double d, double* p);   // return fractional part of d, place integral part in *p
double frexp(double d, int* p);     // find x in [.5,1) and y so that d = x*pow(2,y),
                                    // return x and store y in *p
double fmod(double d, double m);    // floating-point remainder, same sign as d
double ldexp(double d, int i);      // d*pow(2,i)
```

In addition, <*cmath*> and <*math.h*> supply these functions for ***float*** and ***long double*** arguments.

Where several values are possible results – as with ***asin()*** – the one nearest to ***0*** is returned. The result of ***acos()*** is non-negative.

Errors are reported by setting ***errno*** from <*cerrno*> to ***EDOM*** for a domain error and to ***ERANGE*** for a range error. For example:

```
void f()
{
    errno = 0; // clear old error state
    sqrt(-1);
    if (errno==EDOM) cerr << "sqrt() not defined for negative argument";
    pow(numeric_limits<double>::max(),2);
    if (errno == ERANGE) cerr << "result of pow() too large to represent as a double";
}
```

For historical reasons, a few mathematical functions are found in the <*cstdlib*> header rather than in <*cmath*>:

```
int abs(int);           // absolute value
long abs(long);         // absolute value (not in C)
long labs(long);        // absolute value

struct div_t { implementation_defined quot, rem; };
struct ldiv_t { implementation_defined quot, rem; };

div_t div(int n, int d);            // divide n by d, return (quotient,remainder)
ldiv_t div(long int n, long int d); // divide n by d, return (quotient,remainder) (not in C)
ldiv_t ldiv(long int n, long int d); // divide n by d, return (quotient,remainder)
```

22.4 Vector Arithmetic [num.valarray]

Much numeric work relies on relatively simple single-dimensional vectors of floating-point values. In particular, such vectors are well supported by high-performance machine architectures, libraries relying on such vectors are in wide use, and very aggressive optimization of code using such vectors is considered essential in many fields. Consequently, the standard library provides a vector – called ***valarray*** – designed specifically for speed of the usual numeric vector operations.

When looking at the ***valarray*** facilities, it is wise to remember that they are intended as a relatively low-level building block for high-performance computation. In particular, the primary design criterion wasn't ease of use, but rather effective use of high-performance computers when relying on aggressive optimization techniques. If your aim is flexibility and generality rather than efficiency, you are probably better off building on the standard containers from Chapter 16 and Chapter 17 than trying to fit into the simple, efficient, and deliberately traditional framework of ***valarray***.

One could argue that ***valarray*** should have been called ***vector*** because it is a traditional mathematical vector and that ***vector*** (§16.3) should have been called ***array***. However, this is not the way the terminology evolved. A ***valarray*** is a vector optimized for numeric computation, a ***vector*** is a flexible container designed for holding and manipulating objects of a wide variety of types, and an array is a low-level, built-in type.

The ***valarray*** type is supported by four auxiliary types for specifying subsets of a ***valarray***:

- ***slice_array*** and ***gslice_array*** represent the notion of slices (§22.4.6, §22.4.8),
- ***mask_array*** specifies a subset by marking each element in or out (§22.4.9), and
- ***indirect_array*** lists the indices of the elements to be considered (§22.4.10).

22.4.1 Valarray Construction [num.valarray.ctor]

The ***valarray*** type and its associated facilities are defined in namespace ***std*** and presented in <*valarray*>:

```
template<class T> class std::valarray {
    // representation
public:
    typedef T value_type;

    valarray();                                  // valarray with size()==0
    explicit valarray(size_t n);                 // n elements with value T()
    valarray(const T& val, size_t n);            // n elements with value val
    valarray(const T* p, size_t n);              // n elements with values p[0], p[1], ...
    valarray(const valarray& v);                 // copy of v

    valarray(const slice_array<T>&);             // see §22.4.6
    valarray(const gslice_array<T>&);            // see §22.4.8
    valarray(const mask_array<T>&);              // see §22.4.9
    valarray(const indirect_array<T>&);          // see §22.4.10

    ~valarray();

    // ...
};
```

This set of constructors allows us to initialize ***valarray***s from the auxiliary numeric array types and from single values. For example:

```
valarray<double> v0;                    // placeholder, we can assign to v0 later
valarray<float> v1(1000);               // 1000 elements with value float()==0.0F

valarray<int> v2(-1,2000);              // 2000 elements with value –1
valarray<double> v3(100,9.8064);        // bad mistake: floating-point valarray size

valarray<double> v4 = v3;               // v4 has v3.size() elements
```

In the two-argument constructors, the value comes before the number of elements. This differs from the convention for other standard containers (§16.3.4).

The number of elements of an argument ***valarray*** to a copy constructor determines the size of the resulting ***valarray***.

Most programs need data from tables or input; this is supported by a constructor that copies elements from a built-in array. For example:

```
const double vd[] = { 0, 1, 2, 3, 4 };
const int vi[] = { 0, 1, 2, 3, 4 };

valarray<double> v3(vd,4);      // 4 elements: 0,1,2,3
valarray<double> v4(vi,4);      // type error: vi is not pointer to double
valarray<double> v5(vd,8);      // undefined: too few elements in initializer
```

This form of initialization is important because numeric software that produces data in the form of large arrays is common.

The ***valarray*** and its auxiliary facilities were designed for high-speed computing. This is reflected in a few constraints on users and by a few liberties granted to implementers. Basically, an implementer of ***valarray*** is allowed to use just about every optimization technique you can think of. For example, operations may be inlined and the ***valarray*** operations are assumed to be free of side effects (except on their explicit arguments of course). Also, ***valarray***s are assumed to be alias free, and the introduction of auxiliary types and the elimination of temporaries is allowed as long as the basic semantics are maintained. Thus, the declarations in **<*valarray*>** may look somewhat different from what you find here (and in the standard), but they should provide the same operations with the same meaning for code that doesn't go out of the way to break the rules. In particular, the elements of a ***valarray*** should have the usual copy semantics (§17.1.4).

22.4.2 Valarray Subscripting and Assignment [num.valarray.sub]

For ***valarray***s, subscripting is used both to access individual elements and to obtain subarrays:

```
template<class T> class valarray {
public:
    // ...
    valarray& operator=(const valarray& v);    // copy v
    valarray& operator=(const T& val);         // assign val to every element

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

```
    valarray operator[](slice) const;                          // see §22.4.6
    slice_array<T> operator[](slice);

    valarray operator[](const gslice&) const;                  // see §22.4.8
    gslice_array<T> operator[](const gslice&);

    valarray operator[](const valarray<bool>&) const;          // see §22.4.9
    mask_array<T> operator[](const valarray<bool>&);

    valarray operator[](const valarray<size_t>&) const;        // see §22.4.10
    indirect_array<T> operator[](const valarray<size_t>&);

    valarray& operator=(const slice_array<T>&);                // see §22.4.6
    valarray& operator=(const gslice_array<T>&);               // see §22.4.8
    valarray& operator=(const mask_array<T>&);                 // see §22.4.9
    valarray& operator=(const indirect_array<T>&);             // see §22.4.10

    // ...
};
```

A ***valarray*** can be assigned to another of the same size. As one would expect, ***v1=v2*** copies every element of ***v2*** into its corresponding position in ***v1***. If ***valarray***s have different sizes, the result of assignment is undefined. Because ***valarray*** is designed to be optimized for speed, it would be unwise to assume that assigning with a ***valarray*** of the wrong size would cause an easily comprehensible error (such as an exception) or other ''reasonable'' behavior.

In addition to this conventional assignment, it is possible to assign a scalar to a ***valarray***. For example, ***v=7*** assigns ***7*** to every element of the ***valarray v***. This may be surprising, and is best understood as an occasionally useful degenerate case of the operator assignment operations (§22.4.3).

Subscripting with an integer behaves conventionally and does not perform range checking.

In addition to the selection of individual elements, ***valarray*** subscripting provides four ways of extracting subarrays (§22.4.6). Conversely, assignment (and constructors §22.4.1) accepts such subarrays as operands. The set of assignments on ***valarray*** ensures that it is not necessary to convert an auxiliary array type, such as ***slice_array***, to ***valarray*** before assigning it. An implementation may similarly replicate other vector operations, such as + and *, to assure efficiency. In addition, many powerful optimization techniques exist for vector operations involving ***slice***s and the other auxiliary vector types.

22.4.3 Member Operations [num.valarray.member]

The obvious, as well as a few less obvious, member functions are provided:

```
template<class T> class valarray {
public:
    // ...

    valarray& operator*=(const T& arg);        // v[i]*=arg for every element
    // similarly: /=, %=, +=, -=, ^=, &=, |=, <<=, and >>=

    T sum() const;                             // sum of elements
```

```
    valarray shift(int i) const;                    // logical shift (left for 0<i, right for i<0)
    valarray cshift(int i) const;                   // cyclic shift (left for 0<i, right for i<0)

    valarray apply(T f(T)) const;                   // result[i] = f(v[i]) for every element
    valarray apply(T f(const T&)) const;

    valarray operator-() const;                     // result[i] = -v[i] for every element
    valarray operator+() const;                     // result[i] = +v[i] for every element
    valarray operator~() const;                     // result[i] = ~v[i] for every element
    valarray operator!() const;                     // result[i] = !v[i] for every element

    T min() const; // smallest value using < for comparison; if size()==0 the value is undefined
    T max() const; // largest value using < for comparison; if size()==0 the value is undefined

    size_t size() const;                            // number of elements
    void resize(size_t n, const T& val = T());      // n elements with value val
};
```

For example, if `v` is a `valarray`, it can be scaled like this: `v*=.2`, and this: `v/=1.3`. That is, applying a scalar to a vector means applying the scalar to each element of the vector. As usual, it is easier to optimize uses of `*=` than uses of a combination of `*` and `=` (§11.3.1).

Note that the non-assignment operations construct a new `valarray`. For example:

```
double incr(double d) { return d+1; }

void f(valarray<double>& v)
{
    valarray<double> v2 = v.apply(incr);        // produce incremented valarray
}
```

This does not change the value of `v`. Unfortunately, `apply()` does not accept a function object (§18.4) as an argument (§22.9[1]).

The logical and cyclic shift functions, `shift()` and `cshift()`, return a new `valarray` with the elements suitably shifted and leave the original one unchanged. For example, the cyclic shift `v2=v.cshift(n)` produces a `valarray` so that `v2[i]==v[(i+n)%v.size()]`. The logical shift `v3=v.shift(n)` produces a `valarray` so that `v3[i]` is `v[i+n]` if `i+n` is a valid index for `v`. Otherwise, the result is the default element value. This implies that both `shift()` and `cshift()` shift left when given a positive argument and right when given a negative argument. For example:

```
void f()
{
    int alpha[] = { 1, 2, 3, 4, 5 ,6, 7, 8 };
    valarray<int> v(alpha,8);           // 1, 2, 3, 4, 5, 6, 7, 8
    valarray<int> v2 = v.shift(2);      // 3, 4, 5, 6, 7, 8, 0, 0
    valarray<int> v3 = v<<2;            // 4, 8, 12, 16, 20, 24, 28, 32
    valarray<int> v4 = v.shift(-2);     // 0, 0, 1, 2, 3, 4, 5, 6
    valarray<int> v5 = v>>2;            // 0, 0, 0, 1, 1, 1, 1, 2
    valarray<int> v6 = v.cshift(2);     // 3, 4, 5, 6, 7, 8, 1, 2
    valarray<int> v7 = v.cshift(-2);    // 7, 8, 1, 2, 3, 4, 5, 6
}
```

For `valarray`s, `>>` and `<<` are bit shift operators, rather than element shift operators or I/O

operators (§22.4.4). Consequently, <<= and >>= can be used to shift bits within elements of an integral type. For example:

```
void f(valarray<int> vi, valarray<double> vd)
{
    vi <<= 2; // vi[i]<<=2 for all elements of vi
    vd <<= 2; // error: shift is not defined for floating-point values
}
```

It is possible to change the size of a ***valarray***. However, ***resize()*** is ***not*** an operation intended to make ***valarray*** into a data structure that can grow dynamically the way a ***vector*** and a ***string*** can. Instead, ***resize()*** is a re-initialize operation that replaces the existing contents of a ***valarray*** by a set of default values. The old values are lost.

Often, a resized ***valarray*** is one that we created as an empty vector. Consider how we might initialize a ***valarray*** from input:

```
void f()
{
    int n = 0;
    cin >> n;                                   // read array size
    if (n<=0) error("bad array bound");

    valarray<double> v(n);                      // make an array of the right size
    int i = 0;
    while (i<n && cin>>v[i++]) ;                // fill array
    if (i!=n) error("too few elements on input");

    // ...
}
```

If we want to handle the input in a separate function, we might do it like this:

```
void initialize_from_input(valarray<double>& v)
{

    int n = 0;
    cin >> n;                                   // read array size
    if (n<=0) error("bad array bound");

    v.resize(n);                                // make v the right size
    int i = 0;
    while (i<n && cin>>v[i++]) ;                // fill array
    if (i!=n) error("too few elements on input");
}

void g()
{
    valarray<double> v;                 // make a default array
    initialize_from_input(v);           // give v the right size and elements
    // ...
}
```

This avoids copying large amounts of data.

If we want a *valarray* holding valuable data to grow dynamically, we must use a temporary:

```
void grow(valarray<int>& v, size_t n)
{
    if (n<=v.size()) return;

    valarray<int> tmp(n);                    // n default elements

    copy(&v[0],&v[v.size()],&tmp[0]);        // copy algorithm from §18.6.1
    v.resize(n);
    copy(&tmp[0],&tmp[v.size()],&v[0]);
}
```

This is *not* the intended way to use *valarray*. A *valarray* is intended to have a fixed size after being given its initial value.

The elements of a *valarray* form a sequence; that is, `v[0]..v[n-1]` are contiguous in memory. This implies that `T*` is a random-access iterator (§19.2.1) for *valarray<T>* so that standard algorithms, such as *copy*(), can be used. However, it would be more in the spirit of *valarray* to express the copy in terms of assignment and subarrays:

```
void grow2(valarray<int>& v, size_t n)
{
    if (n<=v.size()) return;

    valarray<int> tmp(n);      // n default elements
    slice s(0,v.size(),1);     // subarray of v.size() elements (see §22.4.5)

    tmp[s] = v;
    v.resize(n);
    v[s] = tmp;
}
```

If for some reason input data is organized so that you have to count the elements before knowing the size of vector needed to hold them, it is usually best to read the input into a *vector* (§16.3.5) and then copy the elements into a *valarray*.

22.4.4 Nonmember Operations [valarray.ops]

The usual binary operators and mathematical functions are provided:

```
template<class T> valarray<T> operator*(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator*(const valarray<T>&, const T&);
template<class T> valarray<T> operator*(const T&, const valarray<T>&);

// similarly: /, %, +, –, ^, &, |, <<, >>, &&, ||, ==, !=, <, >, <=, >=, atan2, and pow

template<class T> valarray<T> abs(const valarray<T>&);

// similarly: acos, asin, atan, cos, cosh, exp, log, log10, sin, sinh, sqrt, tan, and tanh
```

The binary operations are defined for *valarray*s and for combinations of a *valarray* and its scalar type. For example:

```
void f(valarray<double>& v, valarray<double>& v2, double d)
{
    valarray<double> v3 = v*v2;    // v3[i] = v[i]*v2[i] for all i
    valarray<double> v4 = v*d;     // v4[i] = v[i]*d for all i
    valarray<double> v5 = d*v2;    // v5[i] = d*v2[i] for all i

    valarray<double> v6 = cos(v);  // v6[i] = cos(v[i]) for all i
}
```

These vector operations all apply their operations to each element of their operand(s) in the way indicated by the `*` and ***cos***`()` examples. Naturally, an operation can be used only if the corresponding operation is defined for the template argument type. Otherwise, the compiler will issue an error when trying to specialize the template (§13.5).

Where the result is a ***valarray***, its length is the same as its ***valarray*** operand. If the lengths of the two arrays are not the same, the result of a binary operator on two ***valarray***s is undefined.

Curiously enough, no I/O operations are provided for ***valarray*** (§22.4.3); `<<` and `>>` are shift operations. However, I/O versions of `>>` and `<<` for ***valarray*** are easily defined (§22.9[5]).

Note that these ***valarray*** operations return new ***valarray***s rather than modifying their operands. This can be expensive, but it doesn't have to be when aggressive optimization techniques are applied (e.g., see §22.4.7).

All of the operators and mathematical functions on ***valarray***s can also be applied to ***slice_array***s (§22.4.6), ***gslice_array***s (§22.4.8), ***mask_array***s (§22.4.9), ***indirect_array***s (§22.4.10), and combinations of these types. However, an implementation is allowed to convert an operand that is not a ***valarray*** to a ***valarray*** before performing a required operation.

22.4.5 Slices [num.slice]

A ***slice*** is an abstraction that allows us to manipulate a vector efficiently as a matrix of arbitrary dimension. It is the key notion of Fortran vectors and of the BLAS (Basic Linear Algebra Subprograms) library, which is the basis for much numeric computation. Basically, a slice is every ***n***th element of some part of a ***valarray***:

```
class std::slice {
    // starting index, a length, and a stride
public:
    slice();
    slice(size_t start, size_t size, size_t stride);

    size_t start() const;      // index of first element
    size_t size() const;       // number of elements
    size_t stride() const;     // element n is at start()+n*stride()
};
```

A *stride* is the distance (in number of elements) between two elements of the ***slice***. Thus, a ***slice*** describes a sequence of integers. For example:

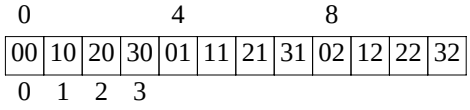
```
size_t slice_index(const slice& s, size_t i) // map i to its corresponding index
{
    return s.start()+i*s.stride();
}

void print_seq(const slice& s)   // print the elements of s
{
    for (int i = 0; i<s.size(); i++) cout << slice_index(s,i) << " ";
}

void f()
{
    print_seq(slice(0,3,4)); // row 0
    cout << ", ";
    print_seq(slice(1,3,4)); // row 1
    cout << ", ";
    print_seq(slice(0,4,1)); // column 0
    cout << ", ";
    print_seq(slice(4,4,1)); // column 1
}
```

prints *0 4 8 , 1 5 9 , 0 1 2 3 , 4 5 6 7*.

In other words, a ***slice*** describes a mapping of non-negative integers into indices. The number of elements (the ***size***( )) doesn't affect the mapping (addressing) but simply allows us to find the end of a sequence. This mapping can be used to simulate two-dimensional arrays within a one-dimensional array (such as ***valarray***) in an efficient, general, and reasonably convenient way. Consider a 3-by-4 matrix the way we often think of it (§C.7):

00	01	02
10	11	12
20	21	22
30	31	32

Following Fortran conventions, we can lay it out in memory like this:



This is *not* the way arrays are laid out in C++ (see §C.7). However, we should be able to present a concept with a clean and logical interface and then choose a representation to suit the constraints of the problem. Here, I have chosen to use Fortran layout to ease the interaction with numeric software that follows that convention. I have not, however, gone so far as to start indexing from *1* rather than *0*; that is left as an exercise (§22.9[9]). Much numeric computation is done and will remain done in a mixture of languages and using a variety of libraries. Often the ability to manipulate data in a variety of formats determined by those libraries and language standards is essential.

Row *x* can be described by a *slice*(*x*,3,4). That is, the first element of row *x* is the *x*th element of the vector, the next element of the row is the (*x*+4)th, etc., and there are 3 elements in each row. In the figures, *slice*(0,3,4) describes the row *00*, *01*, and *02*.

Column *y* can be described by *slice*(4**y*,4,1). That is, the first element of column *y* is the 4**y*th element of the vector, the next element of the column is the (4**y*+1)th, etc., and there are 4 elements in each column. In the figures, *slice*(0,4,1) describes the column *00*, *10*, *20*, and *30*.

In addition to its use for simulating two-dimensional arrays, a *slice* can describe many other sequences. It is a fairly general way of specifying very simple sequences. This notion is explored further in §22.4.8.

One way of thinking of a slice is as an odd kind of iterator: a *slice* allows us to describe a sequence of indices for a *valarray*. We could build a real iterator based on that:

```
template<class T> class Slice_iter {
    valarray<T>* v;
    slice s;
    size_t curr;      // index of current element

    T& ref(size_t i) const { return (*v)[s.start()+i*s.stride()]; }
public:
    Slice_iter(valarray<T>* vv, slice ss) :v(vv), s(ss), curr(0) { }

    Slice_iter end()
    {
        Slice_iter t = *this;
        t.curr = s.size();          // index of last-plus-one element
        return t;
    }

    Slice_iter& operator++() { curr++; return *this; }
    Slice_iter operator++(int) { Slice_iter t = *this; curr++; return t; }

    T& operator[](size_t i) { return ref(curr=i); }      // C style subscript
    T& operator()(size_t i) { return ref(curr=i); }      // Fortran-style subscript
    T& operator*() { return ref(curr); }                 // current element

    // ...
};
```

Since a *slice* has a size, we could even provide range checking. Here, I have taken advantage of *slice*::*size*() to provide an *end*() operation to provide an iterator for the one-past-the-end element of the *valarray*.

Since a *slice* can describe either a row or a column, the *Slice_iter* allows us to traverse a *valarray* by row or by column.

For *Slice_iter* to be useful, ==, !=, and < must be defined:

```
template<class T> bool operator==(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return p.curr==q.curr && p.s.stride()==q.s.stride() && p.s.start()==q.s.start();
}
```

```
template<class T> bool operator!=(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return !(p==q);
}

template<class T> bool operator<(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return p.curr<q.curr && p.s.stride()==q.s.stride() && p.s.start()==q.s.start();
}
```

22.4.6 Slice_array [num.slicearray]

From a ***valarray*** and a ***slice***, we can build something that looks and feels like a ***valarray***, but which is really simply a way of referring to the subset of the array described by the slice. Such a ***slice_array*** is defined like this:

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

    void operator=(const valarray<T>&);
    void operator=(const T& val);                  // assign val to each element

    void operator*=(const valarray<T>& val);       // v[i]*=val for each element
    // similarly: /=, %=, +=, –=, ^=, &=, |=, <<=, >>=

    ~slice_array();
private:
    slice_array();                                 // prevent construction
    slice_array(const slice_array&);               // prevent copying
    slice_array& operator=(const slice_array&);    // prevent copying

    valarray<T>* p;                                // implementation-defined representation
    slice s;
};
```

A user cannot directly create a ***slice_array***. Instead, the user subscripts a ***valarray*** to create a ***slice_array*** for a given slice. Once the ***slice_array*** is initialized, all references to it indirectly go to the ***valarray*** for which it is created. For example, we can create something that represents every second element of an array like this:

```
void f(valarray<double>& d)
{
    slice_array<double>& v_even = d[slice(0,d.size()/2,2)];
    slice_array<double>& v_odd = d[slice(1,d.size()/2,2)];

    v_odd *= 2;          // double every odd element of d
    v_even = 0;          // assign 0 to every even element of d
}
```

The ban on copying ***slice_array***s is necessary so as to allow optimizations that rely on absence of aliases. It can be quite constraining. For example:

```
slice_array<double> row(valarray<double>& d, int i)
{
    slice_array<double> v = d[slice(0,2,d.size()/2)];  // error: attempt to copy

    return d[slice(i%2,i,d.size()/2)];                 // error: attempt to copy
}
```

Often copying a ***slice*** is a reasonable alternative to copying a ***slice_array***.

Slices can be used to express a variety of subsets of an array. For example, we might use slices to manipulate contiguous subarrays like this:

```
inline slice sub_array(size_t first, size_t count) // [first:first+count[
{
    return slice(first,count,1);
}

void f(valarray<double>& v)
{
    size_t sz = v.size();
    if (sz<2) return;
    size_t n = sz/2;
    size_t n2 = sz-n;

    valarray<double> half1(n);
    valarray<double> half2(n2);

    half1 = v[sub_array(0,n)];          // copy of first half of v
    half2 = v[sub_array(n,n2)];         // copy of second half of v

    // ...
}
```

The standard library does not provide a matrix class. Instead, the intent is for ***valarray*** and ***slice*** to provide the tools for building matrices optimized for a variety of needs. Consider how we might implement a simple two-dimensional matrix using a ***valarray*** and ***slice_array***s:

```
class Matrix {
    valarray<double>* v;
    size_t d1, d2;
public:
    Matrix(size_t x, size_t y);          // note: no default constructor
    Matrix& Matrix(const Matrix&);
    Matrix& operator=(const Matrix&);
    ~Matrix();

    size_t size() const { return d1*d2; }
    size_t dim1() const { return d1; }
    size_t dim2() const { return d2; }

    Slice_iter<double> row(size_t i);
    Cslice_iter<double> row(size_t i) const;
```

```
    Slice_iter<double> column(size_t i);
    Cslice_iter<double> column(size_t i) const;

    double& operator()(size_t x, size_t y);                    // Fortran-style subscripts
    double operator()(size_t x, size_t y) const;

    Slice_iter<double> operator()(size_t i) { return row(i); }
    Cslice_iter<double> operator()(size_t i) const { return row(i); }

    Slice_iter<double> operator[](size_t i) { return row(i); }      // C-style subscript
    Cslice_iter<double> operator[](size_t i) const { return row(i); }

    Matrix& operator*=(double);

    valarray<double>& array() { return *v; }
};
```

The representation of a ***Matrix*** is a ***valarray***. We impose dimensionality on that array through slicing. When necessary, we can view that representation as having one, two, three, etc., dimensions in the same way that we provide the default two-dimensional view through ***row()*** and ***column()***. The ***Slice_iter***s are used to circumvent the ban on copying ***slice_arrays***. I couldn't return a ***slice_array***:

```
slice_array<double> row(size_t i) { return (*v)(slice(i,d1,d2)); }
```

so I returned an iterator containing a pointer to the ***valarray*** and the ***slice*** itself instead of a ***slice_array***.

We need an additional class ''iterator for slice of constants,'' ***Cslice_iter*** to express the distinction between a slice of a ***const Matrix*** and a slice of a non-***const Matrix***:

```
inline Slice_iter<double> Matrix::row(size_t i)
{
    return Slice_iter<double>(v,slice(i,d1,d2));
}

inline Cslice_iter<double> Matrix::row(size_t i) const
{
    return Cslice_iter<double>(v,slice(i,d1,d2));
}

inline Slice_iter<double> Matrix::column(size_t i)
{
    return Slice_iter<double>(v,slice(i*d2,d2,1));
}

inline Cslice_iter<double> Matrix::column(size_t i) const
{
    return Cslice_iter<double>(v,slice(i*d2,d2,1));
}
```

The definition of ***Cslice_iter*** is identical to that of ***Slice_iter***, except that it returns ***const*** references to elements of its slice.

The rest of the member operations are fairly trivial:

```
Matrix::Matrix(size_t x, size_t y)
{
    // check that x and y are sensible
    d1 = x;
    d2 = y;
    v = new valarray<double>(x*y);
}

double& Matrix::operator()(size_t x, size_t y)
{
    return row(x)[y];
}

double mul(const valarray<double>& v1, const valarray<double>& v2)
{
    double res = 0;
    for (int i = 0; i<v1.size(); i++) res+= v1[i]*v2[i];
    return res;
}

valarray<double> operator*(const Matrix& m, const valarray<double>& v)
{
    valarray<double> res(m.dim1());
    for (int i = 0; i<m.dim1(); i++) res(i) = mul(m.row(i),v);
    return res;
}

Matrix& Matrix::operator*=(double d)
{
    (*v) *= d;
    return *this;
}
```

I provided `(i,j)` to express ***Matrix*** subscripting because `()` is a single operator and because that notation is the most familiar to many in the numeric community. The concept of a row provides the more familiar (in the C and C++ communities) `[i][j]` notation:

```
void f(Matrix& m)
{
    m(1,2) = 5;          // Fortran-style subscripts
    m.row(1)(2) = 6;
    m.row(1)[2] = 7;
    m[1](2) = 8;         // undesirable mixed style (but it works)
    m[1][2] = 9;         // C++-style subscripts
}
```

The use of ***slice_arrays*** to express subscripting assumes a good optimizer.

Generalizing this to an ***n***-dimensional matrix of arbitrary elements and with a reasonable set of operations is left as an exercise (§22.9[7]).

Maybe your first idea for a two-dimensional vector was something like this:

```
class Matrix {
    valarray< valarray<double> > v;
public:
    // ...
};
```

This would also work (§22.9[10]). However, it is not easy to match the efficiency and compatibility required by high-performance computations without dropping to the lower and more conventional level represented by ***valarray*** plus ***slice***s.

22.4.7 Temporaries, Copying, and Loops [num.matrix]

If you build a vector or a matrix class, you will soon find that three related problems have to be faced to satisfy performance-conscious users:

[1] The number of temporaries must be minimized.

[2] Copying of matrices must be minimized.

[3] Multiple loops over the same data in composite operations must be minimized.

These issues are not directly addressed by the standard library. However, I can outline a technique that can be used to produce highly optimized implementations.

Consider ***U=M*V+W***, where ***U***, ***V***, and ***W*** are vectors and ***M*** is a matrix. A naive implementation introduces temporary vectors for ***M*V*** and ***M*V+W*** and copies the results of ***M*V*** and ***M*V+W***. A smart implementation calls a function ***mul_add_and_assign***(&***U***,&***M***,&***V***,&***W***) that introduces no temporaries, copies no vectors, and touches each element of the matrices the minimum number of times.

This degree of optimization is rarely necessary for more than a few kinds of expressions, so a simple solution to efficiency problems is to provide functions such as ***mul_add_and_assign***() and let the user call those where it matters. However, it is possible to design a ***Matrix*** so that such optimizations are applied automatically for expressions of the right form. That is, we can treat ***U=M*V+W*** as a use of a single operator with four operands. The basic technique was demonstrated for ***ostream*** manipulators (§21.4.6.3). In general, it can be used to make a combination of ***n*** binary operators act like an (***n+1***)-ary operator. Handling ***U=M*V+W*** requires the introduction of two auxiliary classes. However, the technique can result in impressive speedups (say, 30 times) on some systems by enabling more-powerful optimization techniques.

First, we define the result of multiplying a ***Matrix*** by a ***Vector***:

```
struct MVmul {
    const Matrix& m;
    const Vector& v;

    MVmul(const Matrix& mm, const Vector &vv) :m(mm), v(vv) { }

    operator Vector(); // evaluate and return result
};

inline MVmul operator*(const Matrix& mm, const Vector& vv)
{
    return MVmul(mm,vv);
}
```

This ''multiplication'' does nothing except store references to its operands; the evaluation of *M*V* is deferred. The object produced by * is closely related to what is called a ***closure*** in many technical communities. Similarly, we can deal with what happens if we add a *Vector*:

```
struct MVmulVadd {
    const Matrix& m;
    const Vector& v;
    const Vector& v2;

    MVmulVadd(const MVmul& mv, const Vector& vv) :m(mv.m), v(mv.v), v2(vv) { }

    operator Vector(); // evaluate and return result
};

inline MVmulVadd operator+(const MVmul& mv, const Vector& vv)
{
    return MVmulVadd(mv,vv);
}
```

This defers the evaluation of *M*V+W*. We now have to ensure that it all gets evaluated using a good algorithm when it is assigned to a *Vector*:

```
class Vector {
    // ...
public:
    Vector(const MVmulVadd& m)                  // initialize by result of m
    {
        // allocate elements, etc.
        mul_add_and_assign(this,&m.m,&m.v,&m.v2);
    }

    Vector& operator=(const MVmulVadd& m)       // assign the result of m to *this
    {
        mul_add_and_assign(this,&m.m,&m.v,&m.v2);
        return *this;
    }
    // ...
};
```

Now *U=M*V+W* is automatically expanded to

```
U.operator=(MVmulVadd(MVmul(M,V),W))
```

which because of inlining resolves to the desired simple call

```
mul_add_and_assign(&U,&M,&V,&W)
```

Clearly, this eliminates the copying and the temporaries. In addition, we might write *mul_add_and_assign*() in an optimized fashion. However, if we just wrote it in a fairly simple and unoptimized fashion, it would still be in a form that offered great opportunities to an optimizer.

I introduced a new *Vector* (rather than using a *valarray*) because I needed to define assignment (and assignment must be a member function; §11.2.2). However, *valarray* is a strong candidate for the representation of that *Vector*.

The importance of this technique is that most really time-critical vector and matrix computations are done using a few relatively simple syntactic forms. Typically, there is no real gain in optimizing expressions of half-a-dozen operators this way; more conventional techniques (§11.6) suffice.

This technique is based on the idea of using compile-time analysis and closure objects to transfer evaluation of subexpression into an object representing a composite operation. It can be applied to a variety of problems with the common attribute that several pieces of information need to be gathered into one function before evaluation can take place. I refer to the objects generated to defer evaluation as *composition closure objects*, or simply *compositors*.

22.4.8 Generalized Slices [num.gslice]

The ***Matrix*** example in §22.4.6 showed how two ***slice***s could be used to describe rows and columns of a two-dimensional array. In general, a ***slice*** can describe any row or column of an ***n***-dimensional array (§22.9[7]). However, sometimes we need to extract a subarray that is not a row or a column. For example, we might want to extract the 2-by-3 matrix from the top-left corner of a 3-by-4 matrix:

00	01	02
10	11	12
20	21	22
30	31	32

Unfortunately, these elements are not allocated in a way that can be described by a single slice:

0	1	2									
00	10	20	30	01	11	21	31	02	12	22	32
				4	5	6					

A ***gslice*** is a ''generalized slice'' that contains (almost) the information from ***n*** slices:

```
class std::gslice {
    // instead of 1 stride and one size like slice, gslice holds n strides and n sizes
public:
    gslice();
    gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

    size_t start() const;                   // index of first element
    valarray<size_t> size() const;          // number of elements in dimension
    valarray<size_t> stride() const;        // stride for index[0], index[1], ...
};
```

The extra values allow a ***gslice*** to specify a mapping between ***n*** integers and an index to be used to address elements of an array. For example, we can describe the layout of the 2-by-3 matrix by a pair of (length,stride) pairs. As shown in §22.4.5, a length of ***2*** and a stride of ***4*** describes two

elements of a row of the 3-by-4 matrix, when Fortran layout is used. Similarly, a length of *3* and a stride of *1* describes 3 elements of a column. Together, they describe every element of the 2-by-3 submatrix. To list the elements, we can write:

```
size_t gslice_index(const gslice& s, size_t i, size_t j)
{
    return s.start()+i*s.stride()[0]+j*s.stride()[1];
}

size_t len[] = { 2, 3 };        // (len[0],str[0]) describes a row
size_t str[] = { 4, 1 };        // (len[1],str[1]) describes a column

valarray<size_t> lengths(len,2);
valarray<size_t> strides(str,2);

void f()
{
    gslice s(0,lengths,strides);

    for (int i = 0 ; i<s.size()[0]; i++) cout << gslice_index(s,i,0) << " "; // row
    cout << ", ";
    for (int j = 0 ; j<s.size()[1]; j++) cout << gslice_index(s,0,j) << " "; // column
}
```

This prints *0 4 , 0 1 2*.

In this way, a ***gslice*** with two (length,stride) pairs describes a subarray of a 2-dimensional array, a ***gslice*** with three (length,stride) pairs describes a subarray of a 3-dimensional array, etc. Using a ***gslice*** as the index of a ***valarray*** yields a ***gslice_array*** consisting of the elements described by the ***gslice***. For example:

```
void f(valarray<float>& v)
{
    gslice m(0,lengths,strides);
    v[m] = 0; // assign 0 to v[0],v[1],v[2],v[4],v[5],v[6]
}
```

The ***gslice_array*** offers the same set of members as ***slice_array***. In particular, a ***gslice_array*** cannot be constructed directly by the user and cannot be copied (§22.4.6). Instead, a ***gslice_array*** is the result of using a ***gslice*** as the subscript of a ***valarray*** (§22.4.2).

22.4.9 Masks [num.mask]

A ***mask_array*** provides yet another way of specifying a subset of a ***valarray*** and making the result look like a ***valarray***. In the context of ***valarray***s, a mask is simply a ***valarray<bool>***. When a mask is used as a subscript for a ***valarray***, a ***true*** bit indicates that the corresponding element of the ***valarray*** is considered part of the result. This allows us to operate on a subset of a ***valarray*** even if there is no simple pattern (such as a ***slice***) that describes that subset. For example:

```
void f(valarray<double>& v)
{
    bool b[] = { true , false, false, true, false, true };
    valarray<bool> mask(b,6);                  // elements 0, 3, and 5

    valarray<double> vv = cos(v[mask]);        // vv[0]==cos(v[0]), vv[1]==cos(v[3]),
                                               // vv[2]==cos(v[5])
}
```

The ***mask_array*** offers the same set of members as ***slice_array***. In particular, a ***mask_array*** cannot be constructed directly by the user and cannot be copied (§22.4.6). Instead, a ***mask_array*** is the result of using a ***valarray<bool>*** as the subscript of a ***valarray*** (§22.4.2). The number of elements of a ***valarray*** used as a mask must not be greater than the number of elements of the ***valarray*** for which it is used as a subscript.

22.4.10 Indirect Arrays [num.indirect]

An ***indirect_array*** provides a way of arbitrarily subsetting and reordering a ***valarray***. For example:

```
void f(valarray<double>& v)
{
    size_t i[] = { 3, 2, 1, 0 };               // first four elements in reverse order
    valarray<size_t> index(i,4);               // elements 3, 2, 1, 0 (in that order)

    valarray<double> vv = log(v[index]);       // vv[0]==log(v[3]), vv[1]==log(v[2]),
                                               // vv[2]==log(v[1]), vv[3]==log(v[0])
}
```

If an index is specified twice, we have referred to an element of a ***valarray*** twice in the same operation. That's exactly the kind of aliasing that ***valarray***s do not allow, so the behavior of an ***indirect_array*** is undefined if an index is repeated.

The ***indirect_array*** offers the same set of members as ***slice_array***. In particular, a an ***indirect_array*** cannot be constructed directly by the user and cannot be copied (§22.4.6). Instead, an ***indirect_array*** is the result of using a ***valarray<size_t>*** as the subscript of a ***valarray*** (§22.4.2). The number of elements of a ***valarray*** used as a subscript must not be greater than the number of elements of the ***valarray*** for which it is used as a subscript.

22.5 Complex Arithmetic [num.complex]

The standard library provides a ***complex*** template along the lines of the ***complex*** class described in §11.3. The library ***complex*** needs to be a template to serve the need for complex numbers based on different scalar types. In particular, specializations are provided for ***complex*** using ***float***, ***double***, and ***long double*** as its scalar type.

The ***complex*** template is defined in namespace ***std*** and presented in **<*complex*>**:

```
template<class T> class std::complex {
    T re, im;
public:
    typedef T value_type;

    complex(const T& r = T(), const T& i = T()) : re(r), im(i) { }
    template<class X> complex(const complex<X>& a) : re(a.re), im(a.im) { }

    T real() const { return re; }
    T imag() const { return im; }

    complex<T>& operator=(const T& z); // assign complex(z,0)
    template<class X> complex<T>& operator=(const complex<X>&);
    // similarly: +=, -=, *=, /=
};
```

The representation and the inline functions are here for illustration. One could – barely – imagine a standard library ***complex*** that used a different representation. Note the use of member templates to ensure initialization and assignment of any ***complex*** type with any other (§13.6.2).

Throughout this book, I have used ***complex*** as a class rather than as a template. This is feasible because I assumed a bit of namespace magic to get the ***complex*** of ***double*** that I usually prefer:

```
typedef std::complex<double> complex;
```

The usual unary and binary operators are defined:

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

// similarly: -, *, /, ==, and !=

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

The coordinate functions are provided:

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

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

// construct from polar coordinates (abs(),arg()):
template<class T> complex<T> polar(const T& rho, const T& theta);


template<class T> T abs(const complex<T>&);      // sometimes called rho
template<class T> T arg(const complex<T>&);      // sometimes called theta

template<class T> T norm(const complex<T>&);     // square of abs()
```

The usual set of mathematical functions is provided:

```
template<class T> complex<T> sin(const complex<T>&);
//similarly: sinh, sqrt, tan, tanh, cos, cosh, exp, log, and log10

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

Finally, stream I/O is provided:

```
template<class T, class Ch, class Tr>
basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&, complex<T>&);
template<class T, class Ch, class Tr>
basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&, const complex<T>&);
```

A complex is written out in the format (x,y) and can be read in the formats x, (x), and (x,y) (§21.2.3, §21.3.5). The specializations ***complex<float>***, ***complex<double>***, and ***complex<long double>*** are provided to restrict conversions (§13.6.2) and to provide opportunities for optimized implementations. For example:

```
class complex<double> {
    double re, im;
public:
    typedef double value_type;

    complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
    complex(const complex<float>& a) : re(a.real()), im(a.imag()) { }
    explicit complex(const complex<long double>& a) : re(a.real()), im(a.imag()) { }

    // ...
};
```

Now a ***complex<float>*** can be quietly converted to a ***complex<double>***, while a ***complex< long double>*** can't. Similar specializations ensures that a ***complex<float>*** and a ***complex<double>*** can be quietly converted to a ***complex< long double>*** but that a ***complex< long double>*** cannot be implicitly converted to a ***complex<double>*** or to a ***complex<float>*** and a ***complex<double>*** cannot be implicitly converted to a ***complex<float>***. For example:

```
void f(complex<float> cf, complex<double> cd, complex<long double> cld)
{
    complex<double> c = cf;          // fine
    c = cd;                          // fine
    c = cld;                         // error: possible truncation
    c = complex<double>(cld);        // ok: you asked for truncation

    cf = cld;                        // error: possible truncation
    cf = cd;                         // error: possible truncation
    cf = complex<float>(cld);        // ok: you asked for truncation
    cf = complex<float>(cd);         // ok: you asked for truncation
}
```

22.6 Generalized Numeric Algorithms [num.general]

In <*numeric*>, the standard library provides a few generalized numeric algorithms in the style of the non-numeric algorithms from <*algorithm*> (Chapter 18) :

Generalized Numeric Algorithms <numeric>	
accumulate()	Accumulate results of operation on a sequence
inner_product()	Accumulate results of operation on two sequences
partial_sum()	Generate sequence by operation on a sequence
adjacent_difference()	Generate sequence by operation on a sequence

These algorithms generalize common operations such as computing a sum by letting them apply to all kinds of sequences and by making the operation applied to elements on those sequences a parameter. For each algorithm, the general version is supplemented by a version applying the most common operator for that algorithm.

22.6.1 Accumulate [num.accumulate]

The *accumulate*( ) algorithm can be understood as the generalization of a sum of the elements of a vector. The *accumulate*( ) algorithm is defined in namespace *std* and presented in <*numeric*>:

```
template <class In, class T> T accumulate(In first, In last, T init)
{
    while (first != last) init = init + *first++;   // plus
    return init;
}

template <class In, class T, class BinOp> T accumulate(In first, In last, T init, BinOp op)
{
    while (first != last) init = op(init, *first++);      // general operation
    return init;
}
```

The simple version of *accumulate*( ) adds elements of a sequence using their + operator. For example:

```
void f(vector<int>& price, list<float>& incr)
{
    int i = accumulate(price.begin(), price.end(), 0);    // accumulate in int
    double d = 0;
    d = accumulate(incr.begin(), incr.end(), d);          // accumulate in double
    // ...
}
```

Note how the type of the initial value passed determines the return type.

Not all items that we want to add are available as elements of a sequence. Where they are not, we can often supply an operation for *accumulate*( ) to call in order to produce the items to be added. The most obvious kind of operation to pass is one that extracts a value from a data structure. For example:

```
struct Record {
        // ...
        int unit_price;
        int number_of_units;
};

long price(long val, const Record& r)
{
        return val + r.unit_price * r.number_of_units;
}

void f(const vector<Record>& v)
{
        cout << "Total value: " << accumulate(v.begin(),v.end(),0,price) << '\n';
}
```

Operations similar to ***accumulate*** are called ***reduce*** and ***reduction*** in some communities.

22.6.2 Inner_product [num.inner]

Accumulating from a sequence is very common, while accumulating from a pair of sequences is not uncommon. The ***inner_product()*** algorithm is defined in namespace ***std*** and presented in <*numeric*>:

```
template <class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init)
{
        while (first != last) init = init + *first++ * *first2++;
        return init;
}

template <class In, class In2, class T, class BinOp, class BinOp2>
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
        while (first != last) init = op(init,op2(*first++,*first2++));
        return init;
}
```

As usual, only the beginning of the second input sequence is passed as an argument. The second input sequence is assumed to be at least as long as the first.

The key operation in multiplying a ***Matrix*** by a ***valarray*** is an ***inner_product***:

```
valarray<double> operator*(const Matrix& m, const valarray<double>& v)
{
        valarray<double> res(m.dim1());

        for (int i=0; i<m.dim1(); i++) {
                Slice_iter<double>& ri = m.row(i);
                res(i) = inner_product(ri.begin(),ri.end(),&v[0],0);
        }
        return res;
}
```

```
valarray<double> operator*(const valarray<double>& v, const Matrix& m)
{
    valarray<double> res(m.dim2());

    for (int j=0; j<m.dim2(); j++) {
        Slice_iter<double>& cj = m.column(j);
        res(j) = inner_product(&v[0], &v[v.size()], cj.begin(), 0);
    }
    return res;
}
```

Some forms of *inner_product* are often referred to as ''dot product.''

22.6.3 Incremental Change [num.incremental]

The *partial_sum()* and *adjacent_difference()* algorithms are inverses of each other and deal with the notion of incremental change. They are defined in namespace *std* and presented in <*numeric*>:

```
template <class In, class Out> Out adjacent_difference(In first, In last, Out res);

template <class In, class Out, class BinOp>
    Out adjacent_difference(In first, In last, Out res, BinOp op);
```

Given a sequence *a*, *b*, *c*, *d*, etc., *adjacent_difference()* produces *a*, *b*-*a*, *c*-*b*, *d*-*c*, etc.

Consider a vector of temperature readings. We could transform it into a vector of temperature changes like this:

```
vector<double> temps;

void f()
{
    adjacent_difference(temps.begin(), temps.end(), temps.begin());
}
```

For example, *17*, *19*, *20*, *20*, *17* turns into *17*, *2*, *1*, *0*, *-3*.

Conversely, *partial_sum()* allows us to compute the end result of a set of incremental changes:

```
template <class In, class Out, class BinOp>
Out partial_sum(In first, In last, Out res, BinOp op)
{
    if (first==last) return res;
    *res = *first;
    T val = *first;
    while (++first != last) {
        val = op(val, *first);
        *++res = val;
    }
    return ++res;
}
```

```
template<class In, class Out> Out partial_sum(In first, In last, Out res)
{
    return partial_sum(first,last,res,plus);    // §18.4.3
}
```

Given a sequence *a*, *b*, *c*, *d*, etc. , `partial_sum()` produces *a*, *a+b*, *a+b+c*, *a+b+c+d*, etc. For example:

```
void f()
{
    partial_sum(temps.begin(),temps.end(),temps.begin());
}
```

Note the way `partial_sum()` increments `res` before assigning a new value through it. This allows `res` to be the same sequence as its input; `adjacent_difference()` behaves similarly. Thus,

```
partial_sum(v.begin(),v.end(),v.begin());
```

turns the sequence *a*, *b*, *c*, *d* into *a*, *a+b*, *a+b+c*, *a+b+c+d*, and

```
adjacent_difference(v.begin(),v.end(),v.begin());
```

turns it back into the original. In particular, `partial_sum()` turns *17*, *2*, *1*, *0*, *-3* back into *17*, *19*, *20*, *20*, *17*.

For people who think of temperature differences as a boring detail of meteorology or science lab experiments, I point out that analyzing changes in stock prices involves exactly the same two operations.

22.7 Random Numbers [num.random]

Random numbers are essential to many simulations and games. In `<cstdlib>` and `<stdlib.h>`, the standard library provides a simple basis for the generation of random numbers:

```
#define RAND_MAX implementation_defined /* large positive integer */

int rand();          // pseudo-random number between 0 and RAND_MAX
int srand(int i);    // seed random number generator by i
```

Producing a good random-number generator isn't easy, and unfortunately not all systems deliver a good `rand()`. In particular, the low-order bits of a random number are often suspect, so `rand()%n` is not a good portable way of generating a random number between *0* and *n-1*. Often, `(double(rand())/RAND_MAX)*n` gives acceptable results.

A call of `srand()` starts a new sequence of random numbers from the *seed* given as argument. For debugging, it is often important that a sequence of random numbers from a given seed be repeatable. However, we often want to start each real run with a new seed. In fact, to make games unpredictable, it is often useful to pick a seed from the environment of a program. For such programs, some bits from a real-time clock often make a good seed.

If you must write your own random-number generator, be sure to test it carefully (§22.9[14]).

A random-number generator is often more useful if represented as a class. In that way, random-number generators for different distributions are easily built:

```
class Randint {        // uniform distribution in the interval [0,max]
      unsigned long randx;
public:
      Randint(long s = 0) { randx=s; }
      void seed(long s) { randx=s; }

      // magic numbers chosen to use 31 bits of a 32-bit long:

      int abs(int x) { return x&0x7fffffff; }
      static double max() { return 2147483648.0; } // note: a double
      int draw() { return randx = randx*1103515245 + 12345; }

      double fdraw(){ return abs(draw())/max(); }

      int operator()() { return abs(draw()); }
};

class Urand : public Randint { // uniform distribution in the interval [0:n[
      int n;
public:
      Urand(int nn) { n = nn; }

      int operator()() { int r = n*fdraw(); return (r==n) ? n-1 : r; }
};

class Erand : public Randint { // exponential distribution random number generator
      int mean;
public:
      Erand(int m) { mean=m; }
      int operator()() { return -mean * log( (max()-draw())/max() + .5); }
};
```

Here is a simple test:

```
int main()
{
      Urand draw(10);
      map<int,int> bucket;
      for (int i = 0; i< 1000000; i++) bucket[draw()]++;
      for(int j = 0; j<10; j++) cout << bucket[j] << '\n';
}
```

Unless each bucket has approximately the value 10,000, there is a bug somewhere.

These random-number generators are slightly edited versions of what I shipped with the very first C++ library (actually, the first ''C with Classes'' library; §1.4).

22.8 Advice [num.advice]

[1] Numerical problems are often subtle. If you are not 100% certain about the mathematical aspects of a numerical problem, either take expert advice or experiment; §22.1.
[2] Use ***numeric_limits*** to determine properties of built-in types; §22.2.
[3] Specialize ***numeric_limits*** for user-defined scalar types; §22.2.

[4] Use ***valarray*** for numeric computation when run-time efficiency is more important than flexibility with respect to operations and element types; §22.4.
[5] Express operations on part of an array in terms of slices rather than loops; §22.4.6.
[6] Use compositors to gain efficiency through elimination of temporaries and better algorithms; §22.4.7.
[7] Use ***std*::*complex*** for complex arithmetic; §22.5.
[8] You can convert old code that uses a ***complex*** class to use the ***std*::*complex*** template by using a ***typedef***; §22.5.
[9] Consider ***accumulate***(), ***inner_product***(), ***partial_sum***(), and ***adjacent_difference***() before you write a loop to compute a value from a list; §22.6.
[10] Prefer a random-number class for a particular distribution over direct use of ***rand***(); §22.7.
[11] Be careful that your random numbers are sufficiently random; §22.7.

22.9 Exercises [num.exercises]

1. (∗1.5) Write a function that behaves like ***apply***() from §22.4.3, except that it is a nonmember function and accepts function objects.
2. (∗1.5) Write a function that behaves like ***apply***() from §22.4.3 , except that it is a nonmember function, accepts function objects, and modifies its ***valarray*** argument.
3. (∗2) Complete ***Slice_iter*** (§22.4.5). Take special care when defining the destructor.
4. (∗1.5) Rewrite the program from §17.4.1.3 using ***accumulate***().
5. (∗2) Implement I/O operators << and >> for ***valarray***. Implement a ***get_array***() function that creates a ***valarray*** of a size specified as part of the input itself.
6. (∗2.5) Define and implement a three-dimensional matrix with suitable operations.
7. (∗2.5) Define and implement an ***n***-dimensional matrix with suitable operations.
8. (∗2.5) Implement a ***valarray***-like class and implement + and * for it. Compare its performance to the performance of your C++ implementation's ***valarray***. Hint: Include ***x=0.5(x+y)-z*** among your test cases and try it with a variety of sizes for the vectors ***x***, ***y***, and ***z***.
9. (∗3) Implement a Fortran-style array ***Fort_array*** where indices start from ***1*** rather than ***0***.
10. (∗3) Implement ***Matrix*** using a ***valarray*** member as the representation of the elements (rather than a pointer or a reference to a ***valarray***).
11. (∗2.5) Use compositors (§22.4.7) to implement efficient multidimensional subscripting using the [] notation. For example, ***v1[x]***, ***v2[x][y]***, ***v2[x]***, ***v3[x][y][z]***, ***v3[x][y]***, and ***v3[x]*** should all yield the appropriate elements and subarrays using a simple calculation of an index.
12. (∗2) Generalize the idea from the program in §22.7 into a function that, given a generator as an argument, prints a simple graphical representation of its distribution that can be used as a crude visual check of the generator's correctness.
13. (∗1) If ***n*** is an ***int***, what is the distribution of (***double***(***rand***())/***RAND_MAX***)∗***n***?
14. (∗2.5) Plot points in a square output area. The coordinate pairs for the points should be generated by ***Urand***(***N***), where ***N*** is the number of pixels on a side of the output area. What does the output tell you about the distribution of numbers generated by ***Urand***?
15. (∗2) Implement a Normal distribution generator, ***Nrand***.