24

Numerics

"For every complex problem
there is an answer that is
clear, simple, and wrong."

—H. L. Mencken

This chapter is an overview of some fundamental language and library facilities supporting numeric computation. We present the basic problems of size, precision, and truncation. The central part of the chapter is a discussion of multidimensional arrays – both C-style and an *N*-dimensional matrix library. We introduce random numbers as frequently needed for testing, simulation, and games. Finally, we list the standard mathematical functions and briefly introduce the basic functionality of the standard library complex numbers.

24.1 Introduction

For some people, numerics – that is, serious numerical computations – are everything. Many scientists, engineers, and statisticians are in this category. For many people, numerics are sometimes essential. A computer scientist occasionally collaborating with a physicist would be in this category. For most people, a need for numerics – beyond simple arithmetic of integers and floating-point numbers – is rare. The purpose of this chapter is to address language-technical details needed to deal with simple numerical problems. We do not attempt to teach numerical analysis or the finer points of floating-point operations; such topics are far beyond the scope of this book and blend with domain-specific topics in the application areas. Here, we present

- Issues related to the built-in types having fixed size, such as precision and overflow
- Arrays, both the built-in notion of multidimensional arrays and a **Matrix** library that is better suited to numerical computation
- A most basic description of random numbers
- The standard library mathematical functions
- Complex numbers

The emphasis is on the **Matrix** library that makes handling of matrices (multidimensional arrays) trivial.

24.2 Size, precision, and overflow

When we use the built-in types and usual computational techniques, numbers are stored in fixed amounts of memory; that is, the integer types (**int**, **long**, etc.)

are only approximations of the mathematical notion of integers (whole numbers) and the floating-point types (**float**, **double**, etc.) are (only) approximations of the mathematical notion of real numbers. This implies that from a mathematical point of view, some computations are imprecise or wrong. Consider:

```
float x = 1.0/333;
float sum = 0;
for (int i=0; i<333; ++i) sum+=x;
cout << setprecision(15) << sum << "\n";
```

Running this, we do not get 1 as someone might naively expect, but rather

```
0.999999463558197
```

We expected something like that. What we see here is an effect of a rounding error. A floating-point number has only a fixed number of bits, so we can always "fool it" by specifying a computation that requires more bits to represent a result than the hardware provides. For example, the rational number 1/3 cannot be represented exactly as a decimal number (however many decimals we use). Neither can 1/333, so when we add 333 copies of **x** (the machine's best approximation of 1/333 as a **float**), we get something that is slightly different from 1. Whenever we make significant use of floating-point numbers, rounding errors will occur; the only question is whether the error significantly affects the result.

Always check that your results are plausible. When you compute, you must have some notion of what a reasonable result would look like or you could easily get fooled by some "silly bug" or computation error. Be aware of the possibility of rounding errors and if in doubt, consult an expert or read up on numerical techniques.

TRY THIS

Replace **333** in the example with **10** and run the example again. What result would you expect? What result did you get? You have been warned!

The effects of integers being of fixed size can surface more dramatically. The reason is that floating-point numbers are by definition approximations of (real) numbers, so they tend to lose precision (i.e., lose the least significant bits). Integers, on the other hand, tend to overflow (i.e., lose the most significant bits). That tends to make floating-point errors sneaky (and often unnoticed by novices) and integer errors spectacular (and typically hard not to notice). Remember that we prefer errors to manifest themselves early and spectacularly so that we can fix them.

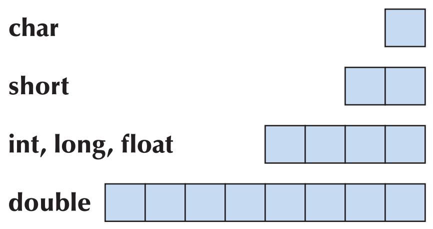
Consider an integer problem:

```
short int y = 40000;
int i = 1000000;
cout << y << "   " << i*i << "\n";
```

Running this, we got the output

```
–25536    –727379968
```

That was expected. What we see here is the effect of overflow. Integer types represent (relatively) small integers only. There just aren't enough bits to exactly represent every number we need in a way that's amenable to efficient computation. Here, a 2-byte **short** integer could not represent 40,000 and a 4-byte **int** can't represent 1,000,000,000,000. The exact sizes of C++ built-in types (§A.8) depend on the hardware and the compiler; **sizeof(x)** gives you the size of **x** in bytes for a variable **x** or a type **x**. By definition, **sizeof(char)==1**. We can illustrate sizes like this:



These sizes are for Windows using a Microsoft compiler. C++ supplies integers and floating-point numbers of a variety of sizes, but unless you have a very good reason for something else, stick to **char**, **int**, and **double**. In most (but of course not all) programs, the remaining integer and floating-point types are more trouble than they are worth.

You can assign an integer to a floating-point variable. If the integer is larger than the floating-point type can represent, you lose precision. For example:

```
cout << "sizes: " << sizeof(int) << ' ' << sizeof(float) << '\n';
int x = 2100000009;     // large int
float f = x;
cout << x << ' ' << f << '\n';
cout << setprecision(15) << x << ' ' << f << '\n';
```

On our machine, this produced

```
Sizes: 4 4
2100000009 2.1e+009
2100000009 2100000000
```

A **float** and an **int** take up the same amount of space (4 bytes). A **float** is represented as a "mantissa" (typically a value between 0 and 1) and an exponent (mantissa*10^{exponent}), so it cannot represent exactly the largest **int**. (If we tried to, where would we find space for the mantissa after we had taken the space needed for the exponent?) As it should, **f** represented **2100000009** as approximately correct as it could. However, that last **9** was too much for it to represent exactly – and that was of course why we chose that number.

On the other hand, when you assign a floating-point number to an integer, you get truncation; that is, the fractional part – the digits after the decimal point – is simply thrown away. For example:

```
float f = 2.8;
int x = f;
cout << x << ' ' << f << '\n';
```

The value of **x** will be **2**. It will not be **3** as you might imagine if you are used to "4/5 rounding rules." C++ **float**-to-**int** conversions truncate rather than round.

When you calculate, you must be aware of possible overflow and truncation. C++ will not catch such problems for you. Consider:

```
void f(int i, double fpd)
{
        char c = i;             // yes: chars really are very small integers
        short s = i;            // beware: an int may not fit in a short int
        i = i+1;                // what if i was the largest int?
        long lg = i*i;          // beware: a long may not be any larger than an int
        float fps = fpd;        // beware: a large double may not fit in a float
        i = fpd;                // truncates: e.g., 5.7 –> 5
        fps = i;                // you can lose precision (for very large int values)
}

void g()
{
        char ch = 0;
        for (int i = 0; i<500; ++i)
                cout << int(ch++) << '\t';
}
```

If in doubt, check, experiment! Don't just despair and don't just read the documentation. Unless you are experienced, it is easy to misunderstand the highly technical documentation related to numerics.

TRY THIS

Run g(). Modify f() to print out c, s, i, etc. Test it with a variety of values.

The representation of integers and their conversions will be examined further in §25.5.3. When we can, we prefer to limit ourselves to a few data types. That can help minimize confusion. For example, by not using float in a program, but only double, we eliminate the possibility of double-to-float conversion problems. In fact, we prefer to limit our use to int, double, and complex (see §24.9) for computation, char for characters, and bool for logical entities. We deal with the rest of the arithmetic types only when we have to.

24.2.1 Numeric limits

In <limits>, <climits>, <limits.h>, and <float.h>, each C++ implementation specifies properties of the built-in types, so that programmers can use those properties to check against limits, set sentinels, etc. These values are listed in §B.9.1 and can be critically important to low-level tool builders. If you think you need them, you are probably too close to the hardware, but there are other uses. For example, it is not uncommon to be curious about aspects of the language implementation, such as "How big is an int?" or "Are chars signed?" Trying to find the definite and correct answers in the system documentation can be difficult, and the standard only specifies minimum requirements. However, a program giving the answer is trivial to write:

```
cout << "number of bytes in an int: " << sizeof(int) << '\n';
cout << "largest int: " << INT_MAX << '\n';
cout << "smallest int value: " << numeric_limits<int>::min() << '\n';

if (numeric_limits<char>::is_signed)
        cout << "char is signed\n";
else
        cout << "char is unsigned\n";

char ch = numeric_limits<char>::min() ;       // smallest positive value
cout << "the char with the smallest positive value: " << ch << '\n';
cout << "the int value of the char with the smallest positive value: "
        << int(ch) << '\n';
```

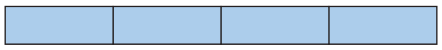
When you write code intended to run on several kinds of hardware, it occasionally becomes immensely valuable to have this kind of information available to the program. The alternative would typically be to hand-code the answers into the program, thereby creating a maintenance hazard.

These limits can also be useful when you want to detect overflow.

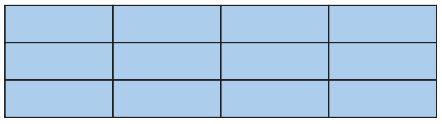
24.3 Arrays

An *array* is a sequence of elements where we can access an element by its index (position). Another word for that general notion is *vector*. Here we are particularly concerned with arrays where the elements are themselves arrays: multidimensional arrays. A common word for a multidimensional array is *matrix*. The variety of names is a sign of the popularity and utility of the general concept. The standard **vector** (§B.4), **array** (§20.9), and the built-in array (§A.8.2) are one-dimensional. So, what if we need two dimensions (e.g., a matrix)? If we need seven dimensions?

We can visualize one- and two-dimensional arrays like this:



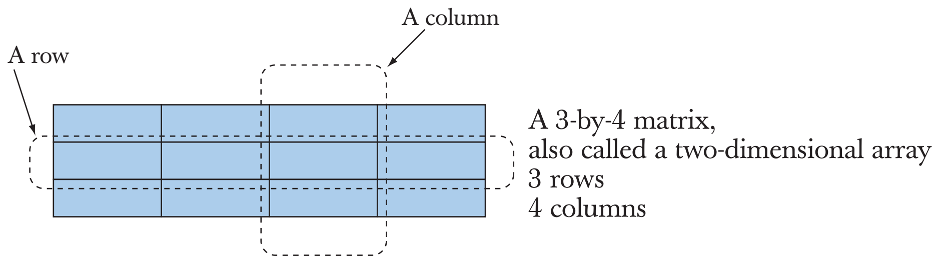
A vector (e.g., **Matrix<int> v(4)**), also called a one-dimensional array, or even a 1-by-*N* matrix



A 3-by-4 matrix (e.g., **Matrix<int,2> m(3,4)**), also called a two-dimensional array

Arrays are fundamental to most computing ("number crunching"). Most interesting scientific, engineering, statistics, and financial computations rely heavily on arrays.

We often refer to an array as consisting of rows and columns:



A column is a sequence of elements with the same first (*x*) coordinate. A row is a set of elements with the same second (*y*) coordinate.

24.4 C-style multidimensional arrays

The C++ built-in array can be used as a multidimensional array. We simply treat a multidimensional array as an array of arrays, that is, an array with arrays as elements. For example:

```
int ai[4];              // 1-dimensional array
double ad[3][4];        // 2-dimensional array
char ac[3][4][5];       // 3-dimensional array
ai[1] = 7;
ad[2][3] = 7.2;
ac[2][3][4] = 'c';
```

This approach inherits the virtues and the disadvantages of the one-dimensional array:

- Advantages
  - Direct mapping to hardware
  - Efficient for low-level operations
  - Direct language support
- Problems
  - C-style multidimensional arrays are arrays of arrays (see below).
  - Fixed sizes (i.e., fixed at compile time). If you want to determine a size at run time, you'll have to use the free store.
  - Can't be passed cleanly. An array turns into a pointer to its first element at the slightest provocation.
  - No range checking. As usual, an array doesn't know its own size.
  - No array operations, not even assignment (copy).

Built-in arrays are widely used for numeric computation. They are also a *major* source of bugs and complexity. For most people, they are a serious pain to write and debug. Look them up if you are forced to use them (e.g., *The C++ Programming Language*). Unfortunately, C++ shares its multidimensional arrays with C, so there is a lot of code "out there" that uses them.

The most fundamental problem is that you can't pass multidimensional arrays cleanly, so you have to fall back on pointers and explicit calculation of locations in a multidimensional array. For example:

```
void f1(int a[3][5]);                  // useful for [3][5] matrices only

void f2(int [ ][5], int dim1);         // 1st dimension can be a variable

void f3(int [5 ][ ], int dim2);        // error: 2nd dimension cannot be a variable

void f4(int[ ][ ], int dim1, int dim2);  // error (and wouldn't work anyway)

void f5(int* m, int dim1, int dim2)      // odd, but works
{
        for (int i=0; i<dim1; ++i)
                for (int j = 0; j<dim2; ++j) m[i*dim2+j] = 0;
}
```

Here, we pass **m** as an **int*** even though it is a two-dimensional array. As long as the second dimension needs to be a variable (a parameter), there really isn't any way of telling the compiler that **m** is a (**dim1**,**dim2**) array, so we just pass a pointer to the start of the memory that holds it. The expression **m[i*dim2+j]** really means **m[i,j]**, but because the compiler doesn't know that **m** is a two-dimensional array, we have to calculate the position of **m[i,j]** in memory.

This is too complicated, primitive, and error-prone for our taste. It can also be slow because calculating the location of an element explicitly complicates optimization. Instead of trying to teach you all about it, we will concentrate on a C++ library that eliminates the problems with the built-in arrays.

24.5 The Matrix library

What are the basic "things" we want from an array/matrix aimed at numerical computation?

- "My code should look very much like what I find in my math/engineering textbook text about arrays."
  - Or about vectors, matrices, tensors.
- Compile-time and run-time checked.
  - Arrays of any dimension.
  - Arrays with any number of elements in a dimension.

- Arrays are proper variables/objects.
  - You can pass them around.
- Usual array operations:
  - Subscripting: **( )**
  - Slicing: **[ ]**
  - Assignment: **=**
  - Scaling operations (**+=**, **–=**, ***=**, **%=**, etc.)
  - Fused vector operations (e.g., **res[i] = a[i]*c+b[i]**)
  - Dot product (**res** = sum of **a[i]*b[i]**; also known as the **inner_product**)
- Basically, transforms conventional array/vector notation into the code you would laboriously have had to write yourself (and runs at least as efficiently as that).
- You can extend it yourself as needed (no "magic" was used in its implementation).

The **Matrix** library does that and only that. If you want more, such as advanced array functions, sparse arrays, control over memory layout, etc., you must write it yourself or (preferably) use a library that better approximates your needs. However, many such needs can be served by building algorithm and data structures on top of **Matrix**. The **Matrix** library is not part of the ISO C++ standard library. You find it on the book support site as **Matrix.h**. It defines its facilities in namespace **Numeric_lib**. We chose the name "matrix" because "vector" and "array" are even more overused in C++ libraries. The plural of *matrix* is *matrices* (with *matrixes* as a rarer form). Where **Matrix** refers to a C++ language entity, we will use **Matrix**es as the plural to avoid confusion. The implementation of the **Matrix** library uses advanced techniques and will not be described here.

24.5.1 Dimensions and access

Consider a simple example:

```
#include "Matrix.h"
using namespace Numeric_lib;

void f(int n1, int n2, int n3)
{
        Matrix<double,1> ad1(n1);       // elements are doubles; one dimension
        Matrix<int,1> ai1(n1);          // elements are ints; one dimension
        ad1(7) = 0;                     // subscript using ( ) — Fortran style
        ad1[7] = 8;                     // [ ] also works — C style
        Matrix<double,2> ad2(n1,n2);          // 2-dimensional
        Matrix<double,3> ad3(n1,n2,n3);       // 3-dimensional
        ad2(3,4) = 7.5;                       // true multidimensional subscripting
        ad3(3,4,5) = 9.2;
}
```

So, when you define a **Matrix** (an object of a **Matrix** class), you specify the element type and the number of dimensions. Obviously, **Matrix** is a template, and the element type and the number of dimensions are template parameters. The result of giving a pair of arguments to **Matrix** (e.g., **Matrix<double,2>**) is a type (a class) of which you can define objects by supplying arguments (e.g., **Matrix<double,2> ad2(n1,n2)**); those arguments specify the dimensions. So, **ad2** is a two-dimensional array with dimensions **n1** and **n2**, also known as an **n1**-by-**n2** matrix. To get an element of the declared element type from a one-dimensional **Matrix**, you subscript with one index; to get an element of the declared element type from a two-dimensional **Matrix**, you subscript with two indices; and so on.

Like built-in arrays, and **vector**s, our **Matrix** indices are zero-based (rather than 1-based like Fortran arrays); that is, the elements of a **Matrix** are numbered [0,max), where max is the number of elements.

This is simple and "straight out of the textbook." If you have problems with this, you need to look at an appropriate math textbook, not a programmer's manual. The only "cleverness" here is that you can leave out the number of dimensions for a **Matrix**: "one-dimensional" is the default. Note also that we can use **[ ]** for subscripting (C and C++ style) or **( )** for subscripting (Fortran style). Having both allows us to better deal with multiple dimensions. The **[x]** subscript notation always takes a single subscript, yielding the appropriate row of the **Matrix**; if **a** is an *N*-dimensional **Matrix**, **a[x]** is an *N*–1-dimensional **Matrix**. The **(x,y,z)** subscript notation takes one or more subscripts, yielding the appropriate element of the **Matrix**; the number of subscripts must equal the number of dimensions.

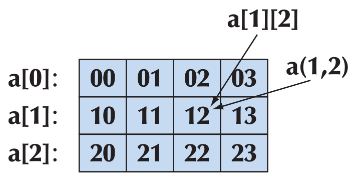
Let's see what happens when we make mistakes:

```
void f(int n1, int n2, int n3)
{
        Matrix<int,0> ai0;          // error: no 0D matrices

        Matrix<double,1> ad1(5);
        Matrix<int,1> ai(5);
        Matrix<double,1> ad11(7);

        ad1(7) = 0;                 // Matrix_error exception (7 is out of range)
        ad1 = ai;                   // error: different element types
        ad1 = ad11;                 // Matrix_error exception (different dimensions)
```

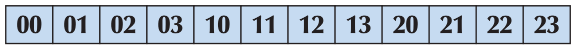
```
    Matrix<double,2> ad2(n1);       // error: length of 2nd dimension missing
    ad2(3) = 7.5;                   // error: wrong number of subscripts
    ad2(1,2,3) = 7.5;               // error: wrong number of subscripts

    Matrix<double,3> ad3(n1,n2,n3);
    Matrix<double,3> ad33(n1,n2,n3);
    ad3 = ad33;              // OK: same element type, same dimensions
}
```

We catch mismatches between the declared number of dimensions and their use at compile time. Range errors we catch at run time and throw a **Matrix_error** exception.

The first dimension is the row and the second the column, so we index a 2D matrix (two-dimensional array) with (row,column). We can also use the [row][column] notation because subscripting a 2D matrix with a single index gives the 1D matrix that is the row. We can visualize that like this:



This **Matrix** will be laid out in memory in "row-first" order:



A **Matrix** "knows" its dimensions, so we can address the elements of a **Matrix** passed as an argument very simply:

```
void init(Matrix<int,2>& a)    // initialize each element to a characteristic value
{
    for (int i=0; i<a.dim1(); ++i)
        for (int j = 0; j<a.dim2(); ++j)
            a(i,j) = 10*i+j;
}

void print(const Matrix<int,2>& a)    // print the elements row by row
{
    for (int i=0; i<a.dim1(); ++i) {
        for (int j = 0; j<a.dim2(); ++j)
            cout << a(i,j) <<'\t';
```

```
                cout << '\n';
            }
    }
```

So, **dim1()** is the number of elements in the first dimension, **dim2()** the number of elements in the second dimension, and so on. The type of the elements and the number of dimensions are part of the **Matrix** type, so we cannot write a function that takes any **Matrix** as an argument (but we could write a template to do that):

```
void init(Matrix& a);    // error: element type and number of dimensions missing
```

Note that the **Matrix** library doesn't supply matrix operations, such as adding two 4D **Matrix**es or multiplying a 2D **Matrix** with a 1D **Matrix**. Doing so elegantly and efficiently is currently beyond the scope of this library. Matrix libraries of a variety of designs could be built on top of the **Matrix** library (see exercise 12).

24.5.2 1D Matrix

What can we do to the simplest **Matrix**, the 1D (one-dimensional) **Matrix**?

We can leave the number of dimensions out of a declaration because 1D is the default:

```
Matrix<int,1> a1(8);     // a1 is a 1D Matrix of ints
Matrix<int> a(8);        // means Matrix<int,1> a(8);
```

So, **a** and **a1** are of the same type (**Matrix<int,1>**). We can ask for the size (the number of elements) and the dimension (the number of elements in a dimension). For a 1D **Matrix**, those are the same.

```
a.size();                // number of elements in Matrix
a.dim1();                // number of elements in 1st dimension
```

We can ask for the elements as laid out in memory, that is, a pointer to the first element:

```
int* p = a.data();       // extract data as a pointer to an array
```

This is useful for passing **Matrix** data to C-style functions taking pointer arguments. We can subscript:

```
a(i);       // ith element (Fortran style), but range checked
a[i];       // ith element (C style), range checked
a(1,2);     // error: a is a 1D Matrix
```

It is common for algorithms to refer to part of a **Matrix**. Such a "part" is called a **slice()** (a sub-**Matrix** or a range of elements) and we provide two versions:

```
a.slice(i);      // the elements from a[i] to the last
a.slice(i,n);    // the n elements from a[i] to a[i+n–1]
```

Subscripts and slices can be used on the left-hand side of an assignment as well as on the right. They refer to the elements of their **Matrix** without making copies of them. For example:

```
a.slice(4,4) = a.slice(0,4);    // assign first half of a to second half
```

For example, if **a** starts out as

```
{ 1 2 3 4 5 6 7 8 }
```

we get

```
{ 1 2 3 4 1 2 3 4 }
```

Note that the most common slices are the "initial elements" of a **Matrix** and the "last elements"; that is, **a.slice(0,j)** is the range [**0**:**j**) and **a.slice(j)** is the range [**j**:**a.size()**). In particular, the example above is most easily written

```
a.slice(4) = a.slice(0,4);      // assign first half of a to second half
```

That is, the notation favors the common cases. You can specify **i** and **n** so that **a.slice(i,n)** is outside the range of **a**. However, the resulting slice will refer only to the elements actually in **a**. For example, **a.slice(i,a.size())** refers to the range [**i**:**a.size()**), and **a.slice(a.size())** and **a.slice(a.size(),2)** are empty **Matrix**es. This happens to be a useful convention for many algorithms. We borrowed that convention from math. Obviously, **a.slice(i,0)** is an empty **Matrix**. We wouldn't write that deliberately, but there are algorithms that are simpler if **a.slice(i,n)** where **n** happens to be **0** is an empty **Matrix** (rather than an error we have to avoid).

We have the usual (for C++ objects) copy operations that copy all elements:

```
Matrix<int> a2 = a;     // copy initialization
a = a2;                 // copy assignment
```

We can apply a built-in operation to each element of a **Matrix**:

```
a *= 7;                 // scaling: a[i]*=7 for each i (also +=, –=, /=, etc.)
a = 7;                  // a[i]=7 for each i
```

This works for every assignment and every composite assignment operator (**=**, **+=**, **–=**, **/=**, ***=**, **%=**, **^=**, **&=**, **|=**, **>>=**, **<<=**) provided the element type supports that operator. We can also apply a function to each element of a **Matrix**:

```
a.apply(f);        // a[i]=f(a[i]) for each element a[i]
a.apply(f,7);      // a[i]=f(a[i],7) for each element a[i]
```

The composite assignment operators and **apply()** modify the elements of their **Matrix** argument. If we instead want to create a new **Matrix** as the result, we can use

```
b = apply(abs,a);    // make a new Matrix with b(i)==abs(a(i))
```

This **abs** is the standard library's absolute value function (§24.8). Basically, **apply(f,x)** relates to **x.apply(f)** as **+** relates to **+=**. For example:

```
b = a*7;           // b[i] = a[i]*7 for each i
a *= 7;            // a[i] = a[i]*7 for each i
y = apply(f,x);    // y[i] = f(x[i]) for each i
x.apply(f);        // x[i] = f(x[i]) for each i
```

Here we get **a==b** and **x==y**.

In Fortran, this second **apply** is called a "broadcast" function and is typically written **f(x)** rather than **apply(f,x)**. To make this facility available for every function **f** (rather than just a selected few functions as in Fortran), we need a name for the "broadcast" operation, so we (re)use **apply**.

In addition, to match the two-argument version of the member **apply**, **a.apply(f,x)**, we provide

```
b = apply(f,a,x);          // b[i]=f(a[i],x) for each i
```

For example:

```
double scale(double d, double s) { return d*s; }
b = apply(scale,a,7);      // b[i] = a[i]*7 for each i
```

Note that the "freestanding" **apply()** takes a function that produces a result from its argument; **apply()** then uses that result to initialize the resulting **Matrix**. Typically it does not modify the **Matrix** to which it is applied. The member **apply()** differs in that it takes a function that modifies its argument; that is, it modifies elements of the **Matrix** to which it is applied. For example:

```
void scale_in_place(double& d, double s) { d *= s; }
b.apply(scale_in_place,7);  // b[i] *= 7 for each i
```

We also supply a couple of the most useful functions from traditional numerics libraries:

```
Matrix<int> a3 = scale_and_add(a,8,a2);        // fused multiply and add
int r = dot_product(a3,a);                     // dot product
```

The **scale_and_add()** operation is often referred to as *fused multiply-add* or simply *fma*; its definition is **result(i)=arg1(i)*arg2+arg3(i)** for each **i** in the **Matrix**. The dot product is also known as the **inner_product** and is described in §21.5.3; its definition is **result+=arg1(i)*arg2(i)** for each **i** in the **Matrix** where **result** starts out as **0**.

One-dimensional arrays are very common; you can represent one as a built-in array, a **vector**, or a **Matrix**. You use **Matrix** if you need the matrix operations provided, such as ***=**, or if the **Matrix** has to interact with higher-dimensional **Matrix**es.

You can explain the utility of a library like this as "It matches the math better" or "It saves you from writing all those loops to do things for each element." Either way, the resulting code is significantly shorter and there are fewer opportunities to make mistakes writing it. The **Matrix** operations – such as copy, assignment to all elements, and operations on all elements – save us from reading or writing a loop (and from wondering if we got the loop exactly right).

Matrix supports two constructors for copying data from a built-in array into a **Matrix**. For example:

```
void some_function(double* p, int n)
{
        double val[] = { 1.2, 2.3, 3.4, 4.5 };
        Matrix<double> data{p,n};
        Matrix<double> constants{val};
        // . . .
}
```

These are often useful when we have our data delivered in terms of arrays or **vector**s from parts of a program not using **Matrix**es.

Note that the compiler is able to deduce the number of elements of an initialized array, so we don't have to give the number of elements when we define **constants** – it is **4**. On the other hand, the compiler doesn't know the number of elements given only a pointer, so for **data** we have to specify both the pointer (**p**) and the number of elements (**n**).

24.5.3 2D Matrix

The general idea of the **Matrix** library is that **Matrix**es of different dimensions really are quite similar, except where you need to be specific about dimensions, so most of what we said about a 1D **Matrix** applies to a 2D **Matrix**:

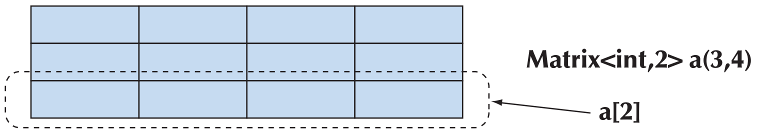
```
Matrix<int,2> a(3,4);

int s = a.size();          // number of elements
int d1 = a.dim1();         // number of elements in a row
int d2 = a.dim2();         // number of elements in a column
int* p = a.data();         // extract data as a pointer to a C-style array
```

We can ask for the total number of elements and the number of elements of each dimension. We can get a pointer to the elements as they are laid out in memory as a matrix.

We can subscript:

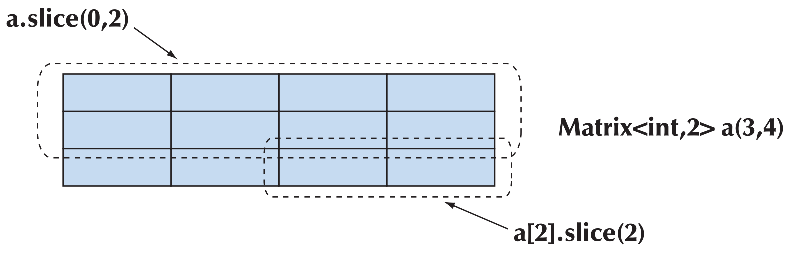
```
a(i,j);                    // (i,j)th element (Fortran style), but range checked
a[i];                      // ith row (C style), range checked
a[i][j];                   // (i,j)th element (C style)
```

For a 2D **Matrix**, subscripting with **[i]** yields the 1D **Matrix** that is the **i**th row. This means that we can extract rows and pass them to operations and functions that require a 1D **Matrix** or even a built-in array (**a[i].data()**). Note that **a(i,j)** may be faster than **a[i][j]**, though that will depend a lot on the compiler and optimizer.



We can take slices:

```
a.slice(i);                // the rows from the a[i] to the last
a.slice(i,n);              // the rows from the a[i] to the a[i+n–1]
```



Note that a slice of a 2D **Matrix** is itself a 2D **Matrix** (possibly with fewer rows).

The distributed operations are the same as for 1D **Matrix**es. These operations don't care how we organize the elements; they just apply to all elements in the order those elements are laid down in memory:

```
Matrix<int,2> a2 = a;     // copy initialization
a = a2;                   // copy assignment
a *= 7;                   // scaling (and +=, –=, /=, etc.)
a.apply(f);               // a(i,j)=f(a(i,j)) for each element a(i,j)
a.apply(f,7);             // a(i,j)=f(a(i,j),7) for each element a(i,j)
b=apply(f,a);             // make a new Matrix with b(i,j)==f(a(i,j))
b=apply(f,a,7);           // make a new Matrix with b(i,j)==f(a(i,j),7)
```

It turns out that swapping rows is often useful, so we supply that:

```
a.swap_rows(1,2);         // swap rows a[1] <–> a[2]
```

There is no **swap_columns()**. If you need it, write it yourself (exercise 11). Because of the row-first layout, rows and columns are not completely symmetrical concepts. This asymmetry also shows up in that **[i]** yields a row (and we have not provided a column selection operator). In that **(i,j)**, the first index, **i**, selects the row. The asymmetry also reflects deep mathematical properties.

There seems to be an infinite number of "things" that are two-dimensional and thus obvious candidates for applications of 2D **Matrix**es:

```
enum Piece { none, pawn, knight, queen, king, bishop, rook };
Matrix<Piece,2> board(8,8);      // a chessboard

const int white_start_row = 0;
const int black_start_row = 7;

Matrix<Piece> start_row
        = {rook, knight, bishop, queen, king, bishop, knight, rook};

Matrix<Piece> clear_row(8) ;     // 8 elements of the default value
```

The initialization of **clear_row** takes advantage of **none==0** and that elements are by default initialized to **0**.

We can use **start_row** and **clear_row** like this:

```
board[white_start_row] = start_row;              // reset white pieces
for (int i = 1; i<7; ++i) board[i] = clear_row;  // clear middle of the board
board[black_start_row] = start_row;              // reset black pieces
```

Note when we extract a row, using **[i]**, we get an lvalue (§4.3); that is, we can assign to the result of **board[i]**.

24.5.4 Matrix I/O

The **Matrix** library provides *very* simple I/O for 1D and 2D **Matrix**es:

```
Matrix<double> a(4);
cin >> a;
cout << a;
```

This will read four whitespace-separated **double**s delimited by curly braces; for example:

```
{ 1.2 3.4 5.6 7.8 }
```

The output is very similar, so that you can read in what you wrote out.

The I/O for 2D **Matrix**es simply reads and writes a curly-brace-delimited sequence of 1D **Matrix**es. For example:

```
Matrix<int,2> m(2,2);
cin >> m;
cout << m;
```

This will read

```
{
{ 1 2 }
{ 3 4 }
}
```

The output will be very similar.

The **Matrix** **<<** and **>>** operators are provided primarily to make the writing of simple programs simple. For more advanced uses, it is likely that you will need to replace them with your own. Consequently, the **Matrix** **<<** and **>>** are placed in the **MatrixIO.h** header (rather than in **Matrix.h**) so that you don't have to include it to use **Matrix**es.

24.5.5 3D Matrix

Basically, a 3D (and higher-dimension) **Matrix** is just like a 2D **Matrix**, except with more dimensions. Consider:

```
Matrix<int,3> a(10,20,30);

a.size();               // number of elements
a.dim1();               // number of elements in dimension 1
a.dim2();               // number of elements in dimension 2
```

```
a.dim3();                 // number of elements in dimension 3
int* p = a.data();        // extract data as a pointer to a C-style array
a(i,j,k);                 // (i,j,k)th element (Fortran style), but range checked
a[i];                     // ith row (C style), range checked
a[i][j][k];               // (i,j,k)th element (C style)
a.slice(i);               // the rows from the ith to the last
a.slice(i,j);             // the rows from the ith to the jth
Matrix<int,3> a2 = a;     // copy initialization
a = a2;                   // copy assignment
a *= 7;                   // scaling (and +=, -=, /=, etc.)
a.apply(f);               // a(i,j,k)=f(a(i,j,k)) for each element a(i,j,k)
a.apply(f,7);             // a(i,j,k)=f(a(i,j,k),7) for each element a(i,j,k)
b=apply(f,a);             // make a new Matrix with b(i,j,k)==f(a(i,j,k))
b=apply(f,a,7);           // make a new Matrix with b(i,j,k)==f(a(i,j,k),7)
a.swap_rows(7,9);         // swap rows a[7] <-> a[9]
```

If you understand 2D **Matrix**es, you understand 3D **Matrix**es. For example, here **a** is 3D, so **a[i]** is 2D (provided **i** is in range), **a[i][j]** is 1D (provided **j** is in range), and **a[i][j][k]** is the **int** element (provided **k** is in range).

We tend to see the world as three-dimensional. That leads to obvious uses of 3D **Matrix**es in modeling (e.g., a physics simulation using a Cartesian grid):

```
int grid_nx;              // grid resolution; set at startup
int grid_ny;
int grid_nz;
Matrix<double,3> cube(grid_nx, grid_ny, grid_nz);
```

And then if we add time as a fourth dimension, we get a 4D space needing a 4D **Matrix**. And so on.

For a more advanced version of **Matrix**, supporting general *N*-dimensional matrices, see Chapter 29 of *The C++ Programming Language*.

24.6 An example: solving linear equations

The code for a numerical computation makes sense if you understand the math that it expresses and tends to appear to be utter nonsense if you don't. The example used here should be rather trivial if you have learned basic linear algebra; if not, just see it as an example of transcribing a textbook solution into code with minimal rewording.

The example here is chosen to demonstrate a reasonably realistic and important use of **Matrix**es. We will solve a set (any set) of linear equations of this form:

$$\begin{gathered} a_{1,1}x_1 + \cdots + a_{1,n}x_n = b_1 \\ \vdots \\ a_{n,1}x_1 + \cdots + a_{n,n}x_n = b_n \end{gathered}$$

Here, the *x*s designate the *n* unknowns; *a*s and *b*s are given constants. For simplicity, we assume that the unknowns and the constants are floating-point values. The goal is to find values for the unknowns that simultaneously satisfy the *n* equations. These equations can compactly be expressed in terms of a matrix and two vectors:

$$\mathbf{Ax} = \mathbf{b}$$

Here, **A** is the square *n*-by-*n* matrix defined by the coefficients:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

The vectors **x** and **b** are the vectors of unknowns and constants, respectively:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

This system may have zero, one, or an infinite number of solutions, depending on the coefficients of the matrix **A** and the vector **b**. There are various methods for solving linear systems. We use a classic scheme, called Gaussian elimination (see Freeman and Phillips, *Parallel Numerical Algorithms*; Stewart, *Matrix Algorithms, Volume I*; and Wood, *Introduction to Numerical Analysis*). First, we transform **A** and **b** so that **A** is an upper-triangular matrix. By upper-triangular, we mean all the coefficients below the diagonal of **A** are zero. In other words, the system looks like this:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & \ddots & \vdots \\ 0 & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

This is easily done. A zero for position $a(i,j)$ is obtained by multiplying the equation for row *i* by a constant so that $a(i,j)$ equals another element in column *j*, say

$a(k,j)$. That done, we just subtract the two equations and $a(i,j) == 0$ and the other values in row i change appropriately.

If we can get all the diagonal coefficients to be nonzero, then the system has a unique solution, which can be found by “back substitution.” The last equation is easily solved:

$$a_{n,n}x_n = b_n$$

Obviously, $x[n]$ is $b[n]/a(n,n)$. That done, eliminate row n from the system and proceed to find the value of $x[n–1]$, and so on, until the value for $x[1]$ is computed. For each n, we divide by $a(n,n)$ so the diagonal values must be nonzero. If that does not hold, the back substitution method fails, meaning that the system has zero or an infinite number of solutions.

24.6.1 Classical Gaussian elimination

Now let us look at the C++ code to express this. First, we’ll simplify our notation by conventionally naming the two **Matrix** types that we are going to use:

```
using Matrix = Numeric_lib::Matrix<double,2> ;
using Vector = Numeric_lib::Matrix<double,1> ;
```

Next we will express our desired computation:

```
Vector classical_gaussian_elimination(Matrix A, Vector b)
{
        classical_elimination(A, b);
        return back_substitution(A, b);
}
```

That is, we make copies of our inputs **A** and **b** (using call by value), call a function to solve the system, and then calculate the result to return by back substitution. The point is that our breakdown of the problem and our notation for the solution are right out of the textbook. To complete our solution, we have to implement **classical_elimination()** and **back_substitution()**. Again, the solution is in the textbook:

```
void classical_elimination(Matrix& A, Vector& b)
{
        const Index n = A.dim1();

        // traverse from 1st column to the next-to-last
        // filling zeros into all elements under the diagonal:
```

```
        for (Index j = 0; j<n-1; ++j) {
                const double pivot = A(j,j);
                if (pivot == 0) throw Elim_failure(j);

                // fill zeros into each element under the diagonal of the ith row:
                for (Index i = j+1; i<n; ++i) {
                        const double mult = A(i,j) / pivot;
                        A[i].slice(j) = scale_and_add(A[j].slice(j), –mult, A[i].slice(j));
                        b(i) –= mult*b(j);          // make the corresponding change to b
                }
        }
}
```

The "pivot" is the element that lies on the diagonal of the row we are currently dealing with. It must be nonzero because we need to divide by it; if it is zero we give up by throwing an exception:

```
Vector back_substitution(const Matrix& A, const Vector& b)
{
        const Index n = A.dim1();
        Vector x(n);

        for (Index i = n–1; i>= 0; ––i) {
                double s = b(i)–dot_product(A[i].slice(i+1),x.slice(i+1));

                if (double m = A(i,i))
                        x(i) = s/m;
                else
                        throw Back_subst_failure(i);
        }

        return x;
}
```

24.6.2 Pivoting

We can avoid the divide-by-zero problem and also achieve a more robust solution by sorting the rows to get zeros and small values away from the diagonal. By "more robust" we mean less sensitive to rounding errors. However, the values change as we go along placing zeros under the diagonal, so we have to also reorder to get small values away from the diagonal (that is, we can't just reorder the matrix and then use the classical algorithm):

```
void elim_with_partial_pivot(Matrix& A, Vector& b)
{
        const Index n = A.dim1();

        for (Index j = 0; j<n; ++j) {
                Index pivot_row = j;

                // look for a suitable pivot:
                for (Index k = j+1; k<n; ++k)
                        if (abs(A(k,j)) > abs(A(pivot_row,j))) pivot_row = k;

                // swap the rows if we found a better pivot:
                if (pivot_row!=j) {
                        A.swap_rows(j,pivot_row);
                        std::swap(b(j), b(pivot_row));
                }

                // elimination:
                for (Index i = j+1; i<n; ++i) {
                        const double pivot = A(j,j);
                        if (pivot==0) error("can't solve: pivot==0");
                        const double mult = A(i,j)/pivot;
                        A[i].slice(j) = scale_and_add(A[j].slice(j), –mult, A[i].slice(j));
                        b(i) –= mult*b(j);
                }
        }
}
```

We use **swap_rows()** and **scale_and_add()** to make the code more conventional and to save us from writing an explicit loop.

24.6.3 Testing

Obviously, we have to test our code. Fortunately, there is a simple way to do that:

```
void solve_random_system(Index n)
{
        Matrix A = random_matrix(n);           // see §24.7
        Vector b = random_vector(n);
        cout << "A = " << A << '\n';
        cout << "b = " << b << '\n';

        try {
                Vector x = classical_gaussian_elimination(A, b);
                cout << "classical elim solution is x = " << x << '\n';
                Vector v = A*x;
                cout << " A*x = " << v << '\n';
        }
        catch(const exception& e) {
                cerr << e.what() << '\n';
        }
}
```

We can get to the **catch** clause in three ways:

- A bug in the code (but, being optimists, we don't think there are any)
- An input that trips up **classical_elimination** (**elim_with_partial_pivot** could do better in many cases)
- Rounding errors

However, our test is not as realistic as we'd like because genuinely random matrices are unlikely to cause problems for **classical_elimination**.

To verify our solution, we print out **A*x**, which had better equal **b** (or close enough for our purpose, given rounding errors). The likelihood of rounding errors is the reason we didn't just do

```
if (A*x!=b) error("substitution failed");
```

Because floating-point numbers are just approximations to real numbers, we have to accept approximately correct answers. In general, using **==** and **!=** on the result of a floating-point computation is best avoided: floating point is inherently an approximation.

The **Matrix** library doesn't define multiplication of a matrix with a vector, so we did that for this program:

```
Vector operator*(const Matrix& m, const Vector& u)
{
        const Index n = m.dim1();
        Vector v(n);
        for (Index i = 0; i<n; ++i) v(i) = dot_product(m[i],u);
        return v;
}
```

Again, a simple **Matrix** operation did most of the work for us. The **Matrix** output operations came from **MatrixIO.h** as described in §24.5.4. The **random_matrix()** and **random_vector()** functions are simple uses of random numbers (§24.7) and are left as an exercise. **Index** is a type alias (§A.16) for the index type used by the **Matrix** library. We brought it into scope with a **using** declaration:

```
using Numeric_lib::Index;
```

24.7 Random numbers

If you ask people for a random number, most say 7 or 17, so it has been suggested that those are the "most random" numbers. People essentially never give the answer 0. Zero is seen to be such a nice round number that it is not perceived as "random" and could therefore be deemed the "least random" number. From a mathematical point of view this is utter nonsense: it is not an individual number that is random. What we often need, and what we often refer to as random numbers, is a sequence of numbers that conform to some distribution and where you cannot easily predict the next number in the sequence given the previous ones. Such numbers are most useful in testing (that's one way of generating a lot of test cases), in games (that is one way of making sure that the next run of the game differs from the previous run), and in simulations (we can make a simulated entity behave in a "random" fashion within the limits of its parameters).

As a practical tool and a mathematical problem, random numbers reach a high degree of sophistication to match their real-world importance. Here, we will just touch the basics as needed for simple testing and simulation. In **<random>**, the standard library provides a sophisticated set of facilities for generating random numbers to match a variety of mathematical distributions. The standard library random number facilities are based on two fundamental notions:

- *Engines* (random number engines): An engine is a function object that generates a uniformly distributed sequence of integer values.
- *Distributions:* A distribution is a function object that generates a sequence of values according to a mathematical formula given a sequence of values from an engine as inputs.

For example, consider the function **random_vector()** that was used in §24.6.3. A call **random_vector(n)** produces a **Matrix<double,1>** with **n** elements of type **double** with random values in the range [**0**:**n**):

```
Vector random_vector(Index n)
{
        Vector v(n);
        default_random_engine ran{};                   // generates integers
        uniform_real_distribution<> ureal{0,max};      // maps ints into doubles
                                                       // in [0:max)
        for (Index i = 0; i < n; ++i)
                v(i) = ureal(ran);

        return v;
}
```

The default engine (**default_random_engine**) is simple, cheap to run, and good enough for casual use. For more professional uses, the standard library offers a variety of engines with better randomness properties and different running costs. Examples are **linear_congurential_engine**, **mersenne_twister_engine**, and **random_device**. If you want to use those, and in general if you need to do better than the **default_random_engine**, you have a bit of reading to do. To get an idea of the quality of your system's random number generator, do exercise 10.

The two random number generators from **std_lib_facilities.h** were defined as

```
int randint(int min, int max)
{
        static default_random_engine ran;
        return uniform_int_distribution<>{min,max}(ran);
}

int randint(int max)
{
        return randint(0,max);
}
```

These simple functions can be most useful, but just to try something else, let us generate a normal distribution:

```
auto gen = bind(normal_distribution<double>{15,4.0},
                        default_random_engine{});
```

The standard library function **bind()** from **<functional>** constructs a function object that when invoked calls its first argument with its second as the argument. So here, **gen()** returns values according to the normal distribution with its mean at **15** and a standard deviation of **4.0** using the **default_random_engine**. We could use it like this:

```
vector<int> hist(2*15);

for (int i = 0; i < 500; ++i)                   // generate histogram of 500 values
        ++hist[int(round(gen()))];
```

```
for (int i = 0; i != hist.size(); ++i) {        // write out histogram
        cout << i << '\t';
        for (int j = 0; j != hist[i]; ++j)
                cout << '*';
        cout << '\n';
}
```

We got

```
0
1
2
3	**
4	*
5	*****
6	****
7	****
8	******
9	************
10	***************************
11	***************************
12	**************************************
13	**********************************************************
14	***************************************************
15	*******************************************************
16	********************************
17	**************************************************
18	*************************************
19	**********************************
20	**************
21	************
22	************
23	*******
24	******
25	*
26	*
27
28
29
```

The normal distribution is very common and also known as the Gaussian distribution or (for obvious reasons) simply "the bell curve." Other distributions include **bernoulli_distribution**, **exponential_distribution**, and **chi_squared_distribution**. You can find them described in *The C++ Programming Language*. Integer distributions return values in a closed interval [**a**:**b**], whereas real (floating-point) distributions return values in open intervals [**a**:**b**).

By default, an engine (except possibly **random_device**) gives the same sequence each time a program is run. That is most convenient for initial debugging. If we want different sequences from an engine, we need to initialize it with different values. Such initializers are conventionally called "seeds." For example:

```
auto gen1 = bind(uniform_int_distribution<>{0,9},
        default_random_engine{});
auto gen2 = bind(uniform_int_distribution<>{0,9},
        default_random_engine{10});
auto gen3 = bind(uniform_int_distribution<>{0,9},
        default_random_engine{5});
```

To get an unpredictable sequence, people often use the time of day (down to the last nanosecond; §26.6.1) or something like that as the seed.

24.8 The standard mathematical functions

The standard mathematical functions (**cos**, **sin**, **log**, etc.) are provided by the standard library. Their declarations are found in **<cmath>**.

Standard mathematical functions	
abs(x)	absolute value
ceil(x)	smallest integer **>= x**
floor(x)	largest integer **<= x**
sqrt(x)	square root; **x** must be nonnegative
cos(x)	cosine
sin(x)	sine
tan(x)	tangent
acos(x)	arccosine; result is nonnegative

Standard mathematical functions (*continued*)	
asin(x)	arcsine; result nearest to 0 returned
atan(x)	arctangent
sinh(x)	hyperbolic sine
cosh(x)	hyperbolic cosine
tanh(x)	hyperbolic tangent
exp(x)	base-e exponential
log(x)	natural logarithm, base-e; **x** must be positive
log10(x)	base-10 logarithm

The standard mathematical functions are provided for types **float**, **double**, **long double**, and **complex** (§24.9) arguments. If you do floating-point computations, you'll find these functions useful. If you need more details, documentation is widely available; your online documentation would be a good place to start.

If a standard mathematical function cannot produce a mathematically valid result, it sets the variable **errno**. For example:

```
errno = 0;
double s2 = sqrt(–1);
if (errno) cerr << "something went wrong with something somewhere";
if (errno == EDOM)          // domain error
        cerr << "sqrt() not defined for negative argument";
pow(very_large,2);          // not a good idea
if (errno==ERANGE)          // range error
        cerr << "pow(" << very_large << ",2) too large for a double";
```

If you do serious mathematical computations, you must check **errno** to ensure that it is still **0** after you get your result. If not, something went wrong. Look at your manual or online documentation to see which mathematical functions can set **errno** and which values they use for **errno**.

As indicated in the example, a nonzero **errno** simply means "Something went wrong." It is not uncommon for functions not in the standard library to set **errno** in case of error, so you have to look more carefully at the value of **errno** to get an idea of exactly what went wrong. If you test **errno** immediately after a standard library function and if you made sure that **errno==0** before calling it, you can rely on the values as we did with **EDOM** and **ERANGE** in the example. **EDOM** is set for a domain error (i.e., a problem with the arguments). **ERANGE** is set for a range error (i.e., a problem with the result).

Error handling based on **errno** is somewhat primitive. It dates from the first (1975 vintage) C mathematical functions.

24.9 Complex numbers

Complex numbers are widely used in scientific and engineering computations. We assume that if you need them, you will know about their mathematical properties, so we'll just show you how complex numbers are expressed in the ISO C++ standard library. You find the declaration of complex numbers and their associated standard mathematical functions in **<complex>**:

```
template<class Scalar> class complex {
        // a complex is a pair of scalar values, basically a coordinate pair
        Scalar re, im;
public:
        constexpr complex(const Scalar & r, const Scalar & i) :re(r), im(i) { }
        constexpr complex(const Scalar & r) :re(r),im(Scalar ()) { }
        complex() :re(Scalar ()), im(Scalar ()) { }

        constexpr Scalar real() { return re; }          // real part
        constexpr Scalar imag() { return im; }          // imaginary part

        // operators: = += –= *= /=
};
```

The standard library **complex** is guaranteed to be supported for scalar types **float**, **double**, and **long double**. In addition to the members of **complex** and the standard mathematical functions (§24.8), **<complex>** offers a host of useful operations:

Complex operators	
z1+z2	addition
z1–z2	subtraction
z1*z2	multiplication
z1/z2	division
z1==z2	equality
z1!=z2	inequality
norm(z)	the square of **abs(z)**

Complex operators (*continued*)	
conj(z)	conjugate: if **z** is **{re,im}**, then **conj(z)** is **(re,–im)**
polar(rho,theta)	make a complex given polar coordinates (**rho,theta**)
real(z)	real part
imag(z)	imaginary part
abs(z)	also known as rho
arg(z)	also known as theta
out << z	complex output
in >> z	complex input

Note: **complex** does not provide **<** or **%**.

Use **complex<T>** exactly like a built-in type, such as **double.** For example:

```
using cmplx = complex<double>;  // sometimes complex<double> gets verbose

void f(cmplx z, vector<cmplx>& vc)
{
        cmplx z2 = pow(z,2);
        cmplx z3 = z2*9.3+vc[3];
        cmplx sum = accumulate(vc.begin(), vc.end(), cmplx{});
        // . . .
}
```

Remember that not all operations that we are used to from **int** and **double** are defined for a **complex**. For example:

```
if (z2<z3)        // error: there is no < for complex numbers
```

Note that the representation (layout) of the C++ standard library complex numbers is compatible with their corresponding types in C and Fortran.

24.10 References

Basically, the issues discussed in this chapter, such as rounding errors, **Matrix** operations, and complex arithmetic, are of no interest and make no sense in isolation. We simply describe (some of) the support provided by C++ to people with the need for and knowledge of mathematical concepts and techniques to do numerical computations.

In case you are a bit rusty in those areas or simply curious, we can recommend some information sources:

The MacTutor History of Mathematics archive, http://www-gap.dcs.st-and.ac.uk/~history

- A great link for anyone who likes math or simply needs to use math
- A great link for someone who would like to see the human side of mathematics; for example, who is the only major mathematician to win an Olympic medal?
    - Famous mathematicians: biographies, accomplishments
    - Curio
- Famous curves
- Famous problems
- Mathematical topics
    - Algebra
    - Analysis
    - Numbers and number theory
    - Geometry and topology
    - Mathematical physics
    - Mathematical astronomy
        - The history of mathematics
        - . . .

Freeman, T. L., and Chris Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.

Gullberg, Jan. *Mathematics – From the Birth of Numbers*. W. W. Norton, 1996. ISBN 039304002X. One of the most enjoyable books on basic and useful mathematics. A (rare) math book that you can read for pleasure and also use to look up specific topics, such as matrices.

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1998. ISBN 0201896842.

Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions*. SIAM, 1998. ISBN 0898714141.

Wood, Alistair. *Introduction to Numerical Analysis*. Addison-Wesley, 1999. ISBN 020194291X.

Drill

1. Print the size of a **char**, a **short**, an **int**, a **long**, a **float**, a **double**, an **int***, and a **double*** (use **sizeof**, not **<limits>**).
2. Print out the size as reported by **sizeof** of **Matrix<int> a(10)**, **Matrix<int> b(100)**, **Matrix<double> c(10)**, **Matrix<int,2> d(10,10)**, **Matrix<int,3> e(10,10,10)**.
3. Print out the number of elements of each of the **Matrix**es from 2.
4. Write a program that takes **int**s from **cin** and outputs the **sqrt()** of each **int**, or "no square root" if **sqrt(x)** is illegal for some **x** (i.e., check your **sqrt()** return values).
5. Read ten floating-point values from input and put them into a **Matrix<double>**. **Matrix** has no **push_back()** so be careful to handle an attempt to enter a wrong number of **double**s. Print out the **Matrix**.
6. Compute a multiplication table for **[0,n)*[0,m)** and represent it as a 2D **Matrix**. Take **n** and **m** from **cin** and print out the table nicely (assume that **m** is small enough that the results fit on a line).
7. Read ten **complex<double>**s from **cin** (yes, **cin** supports **>>** for **complex**) and put them into a **Matrix**. Calculate and output the sum of the ten complex numbers.
8. Read six **int**s into a **Matrix<int,2> m(2,3)** and print them out.

Review

1. Who uses numerics?
2. What is precision?
3. What is overflow?
4. What is a common size of a **double**? Of an **int**?
5. How can you detect overflow?
6. Where do you find numeric limits, such as the largest **int**?
7. What is an array? A row? A column?
8. What is a C-style multidimensional array?
9. What are the desirable properties of language support (e.g., a library) for matrix computation?
10. What is a dimension of a matrix?
11. How many dimensions can a matrix have (in theory/math)?
12. What is a slice?
13. What is a broadcast operation? List a few.
14. What is the difference between Fortran-style and C-style subscripting?
15. How do you apply an operation to each element of a matrix? Give examples.

16. What is a fused operation?
17. Define *dot product*.
18. What is linear algebra?
19. What is Gaussian elimination?
20. What is a pivot? (In linear algebra? In "real life"?)
21. What makes a number random?
22. What is a uniform distribution?
23. Where do you find the standard mathematical functions? For which argument types are they defined?
24. What is the imaginary part of a complex number?
25. What is the square root of –1?

Terms

array	Fortran	scaling
C	fused operation	size
column	imaginary	**sizeof**
complex number	**Matrix**	slicing
dimension	multidimensional	subscripting
dot product	random number	uniform distribution
element-wise operation	real	
errno	row	

Exercises

1. The function arguments **f** for **a.apply(f)** and **apply(f,a)** are different. Write a **triple()** function for each and use each to triple the elements of an array **{ 1 2 3 4 5 }**. Define a single **triple()** function that can be used for both **a.apply(triple)** and **apply(triple,a)**. Explain why it could be a bad idea to write every function to be used by **apply()** that way.
2. Do exercise 1 again, but with function objects, rather than functions. Hint: **Matrix.h** contains examples.
3. Expert level only (this cannot be done with the facilities described in this book): Write an **apply(f,a)** that can take a **void (T&)**, a **T (const T&)**, and their function object equivalents. Hint: **Boost::bind**.
4. Get the Gaussian elimination program to work; that is, complete it, get it to compile, and test it with a simple example.
5. Try the Gaussian elimination program with **A=={ {0 1} {1 0} }** and **b=={ 5 6 }** and watch it fail. Then, try **elim_with_partial_pivot()**.
6. In the Gaussian elimination example, replace the vector operations **dot_product()** and **scale_and_add()** with loops. Test, and comment on the clarity of the code.

7. Rewrite the Gaussian elimination program without using the **Matrix** library; that is, use built-in arrays or **vector**s instead of **Matrix**es.
8. Animate the Gaussian elimination.
9. Rewrite the nonmember **apply()** functions to return a **Matrix** of the return type of the function applied; that is, **apply(f,a)** should return a **Matrix<R>** where **R** is the return type of **f**. Warning: The solution requires information about templates not available in this book.
10. How random is your **default_random_engine**? Write a program that takes two integers **n** and **d** as inputs and calls **randint(n) d** times, recording the result. Output the number of draws for each of [**0**:**n**) and "eyeball" how similar the counts are. Try with low values for **n** and with low values for **d** to see if drawing only a few random numbers causes obvious biases.
11. Write a **swap_columns()** to match **swap_rows()** from §24.5.3. Obviously, to do that you have to read and understand some of the existing **Matrix** library code. Don't worry too much about efficiency: it is not possible to get **swap_columns()** to run as fast as **swap_rows()**.
12. Implement

    ```
    Matrix<double> operator*(Matrix<double,2>&,Matrix<double>&);
    ```

    and

    ```
    Matrix<double,N>operator+(Matrix<double,N>&,Matrix<double,N>&)
    ```

    If you need to, look up the mathematical definitions in a textbook.

Postscript

If you don't feel comfortable with mathematics, you probably didn't like this chapter and you'll probably choose a field of work where you are unlikely to need the information presented here. On the other hand, if you do like mathematics, we hope that you appreciate how closely the fundamental concepts of mathematics can be represented in code.