Chapter 24
Numerics

Bjarne Stroustrup

www.stroustrup.com/Programming

Abstract

- This lecture is an overview of the fundamental tools and techniques for numeric computation. For some, numerics are everything. For many, numerics are occasionally essential. Here, we present the basic problems of size, precision, truncation, and error handling in standard mathematical functions. We present multidimensional matrices and the standard library complex numbers.

Overview

- Precision, overflow, sizes, errors
- Matrices
- Random numbers
- Complex numbers

- Please note:
  - Again, we are exploring how to express the fundamental notions of an application domain in code
    - Numerical computation, especially linear algebra
  - We are showing the basic tools only. The computational techniques for using these tools well, say in engineering calculations, you'll have to learn elsewhere
    - That's "domain knowledge" beyond the scope of this presentation

Precision, etc.

- When we use the built-in types and usual computational techniques, numbers are stored in fixed amounts of memory
  - Floating-point numbers are (only) approximations of real numbers

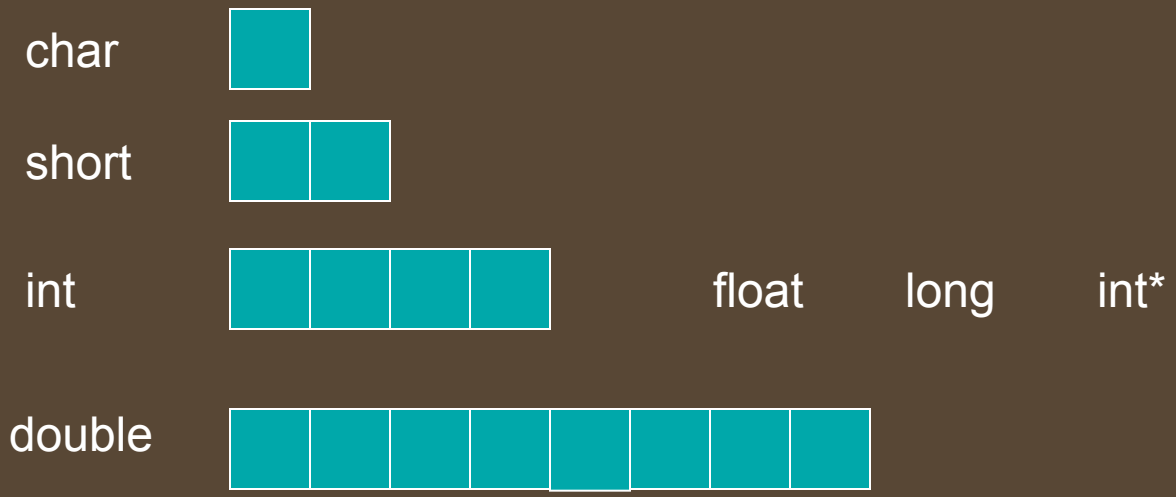
```
float x = 1.0/333;
float sum = 0;
for (int i=0; i<333; ++i)
        sum+=x;
cout << sum << "\n";          // 0.999999
```

  - Integer types represent (relatively) small integers only

```
short y = 40000;
int i = 1000000;
cout << y << "   " << i*i << "\n";          // -25536   -727379968
```

- There just aren't enough bits to exactly represent every number we need in a way that's amenable to efficient computation

Sizes



- The exact sizes of C++ built-in types depends on the hardware and the compiler
  - These sizes are for Windows using Microsoft , GCC on Linux, and others
  - **sizeof(x)** gives you the size of **x** in bytes
  - By definition, **sizeof(char)==1**
- Unless you have a very good reason for something else, stick to **bool**, **char**, **int**, and **double**

Sizes, overflow, truncation

```
// when you calculate, you must be aware of possible overflow and truncation
// Beware: C++ will not catch such problems for you

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

    char ch = 0;    // try this
    for (int i = 0; i<500; ++i)
        { cout << int(ch) << "\t"; ++ch; }}
```

If in doubt, you can check

- The simplest way to test
  - Assign, then compare

```
void f(int i)
{
    char c = i;          // may throw away information you don't want to loose
    if (c != i) {
            // oops! We lost information, figure out what to do
    }
    // …
}
```

  - That's what **narrow_cast** from **std_lib_facilities.h** does

Math function errors

- If a standard mathematical function can't do its job, it sets **errno from <cerrno>**, for example

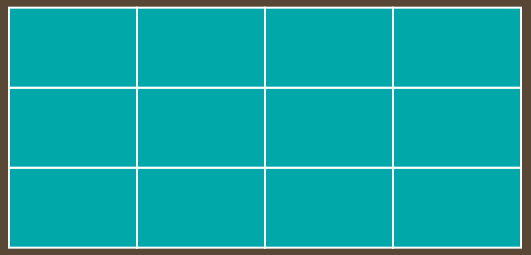
```
void f(double negative, double very_large)
    // primitive (1974 vintage, pre-C++) but ISO standard error handling
{
    errno = 0;                  // no current errors
    sqrt(negative);             // not a good idea
    if (errno)    { /* … */ }   // errno!=0 means "something went wrong"
    if (errno == EDOM)    // domain error
        cerr << "sqrt() not defined for negative argument\n";

    pow(very_large,2);          // not a good idea (and did you spot the bug?)
    if (errno==ERANGE)   // range error
        cerr << "pow(" << very_large << ",2) too large for a double\n";
}
```

Matrices

- The standard **vector** and the built-in array are one dimensional
- What if we need 2 dimensions? (e.g. a matrix)
  - N dimensions?

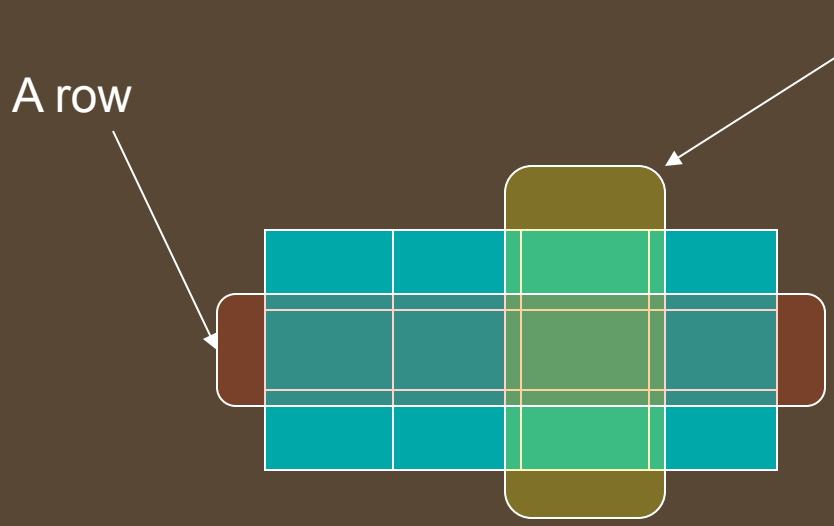
A vector (e.g. **Matrix<int> v(4)**), also called a 1-dimensional Matrix, or even a 1*N matrix



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

Matrices

- **Matrix<int> m(3,4);**



A 3*4 matrix,
also called a 2-dimensional Matrix

3 rows
4 columns

C-style multidimensional Arrays

- A built-in facility

```
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';
```

  - Basically, Arrays of Arrays

C-style multidimensional Arrays

- Problems
  - C-style multidimensional Arrays are Arrays of Arrays
  - Fixed sizes (i.e. fixed at compile time)
    - If you want to determine a size at run-time, you'll have to use free store
  - Can't be passed cleanly
    - An Array turns into a pointer at the slightest provocation
  - No range checking
    - As usual, an Array doesn't know its own size
  - No Array operations
    - Not even assignment (copy)
- A **major** source of bugs
  - And, for most people, they are a serious pain to write
- Look them up **only** if you are forced to use them
  - E.g. TC++PL4, pp. 183-186

C-style multidimensional Arrays

- You can't pass multidimensional Arrays cleanly

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

  - Can't read vector with size from input and then call f1
    - (unless the size happens to be 3*5)
  - Can't write a recursive/adaptive function

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

void f3(int[ ][ ], int dim1, int dim2);  // error (and wouldn't work anyway)

void f4(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;
}
```

A Matrix library

// on the textbook website: Matrix.h

```
#include "Matrix.h"
void f(int n1, int n2, int n3)
{
    Matrix<double> ad1(n1); // default: one dimension
    Matrix<int,1> ai1(n1);
    Matrix<double,2>  ad2(n1,n2);      // 2-dimensional
    Matrix<double,3> ad3(n1,n2,n3);    // 3-dimensional

    ad1(7) = 0;                        // subscript using ( ) – Fortran style
    ad1[7] = 8;                        // [ ] also works – C style

    ad2(3,4) = 7.5;                    // true multidimensional subscripting
    ad3(3,4,5) = 9.2;
}
```

A Matrix library

- "like your math/engineering textbook talks about Matrices"
  - Or about vectors, matrices, tensors
- Compile-time and run-time checked
- Matrices of any dimension
  - 1, 2, and 3 dimensions actually work (you can add more if/as needed)
- Matrices are proper variables/objects
  - You can pass them around
- Usual Matrix 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]**)
- Performs equivalently to hand-written low-level code
- You can extend it yourself as needed ("no magic")

A Matrix library

```
// compile-time and run-time error checking
void f(int n1, int n2, int n3)
{
    Matrix<double> ad1(5);   // default: one dimension
    Matrix<int> ai(5);
    Matrix<double> ad11(7);
    Matrix<double,2>  ad2(n1);          // error: length of 2nd dimension missing
    Matrix<double,3> ad3(n1,n2,n3);
    Matrix<double,3> ad33(n1,n2,n3);

    ad1(7) = 0;                 // Matrix_error exception; 7 is out of range
    ad1 = ai;                   // error: different element types
    ad1 = ad11;                 // Matrix_error exception; different dimensions
    ad2(3) = 7.5;               // error: wrong number of subscripts
    ad3 = ad33;                 // ok: same element type, same dimensions, same
    lengths
}
```

A Matrix library

- As we consider the matrix (row, column):

Matrix<int> a(3,4);

- As the elements are laid out in memory ("row-wise"):

00	01	02	03	10	11	12	13	20	21	22	23

A Matrix library

```
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 of Matrix a, 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';
    }
}
```

2D and 3D Matrices

```
// 2D space (e.g. a game board):
enum Piece { none, pawn, knight, queen, king, bishop, rook };
Matrix<Piece,2> board(8,8);          // a chessboard
Piece init_pos[] = { rook, knight, bishop, queen, king, bishop, knight, rook };

// 3D space (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);
```

1D Matrix

```
Matrix<int> a(10);  // means Matrix<int,1> a(10);
a.size();                       // number of elements
a.dim1();                       // number of elements
int* p = a.data();              // extract data as a pointer to a C-style array
a(i);                           // ith element (Fortran style), but range checked
a[i];                           // ith element (C-style), but range checked

Matrix<int> a2 = a; // copy initialization
a = a2;                         // copy assignment
a *= 7;                         // scaling a(i)*=7 for each i (also +=, -=, /=, etc.)
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)
b =apply(f,a);                  // make a new Matrix with b(i)==f(a(i))
b =apply(f,a,7);                // make a new Matrix with b(i)==f(a(i),7)

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

2D Matrix (very like 1D)

```
Matrix<int,2> a(10,20);
a.size();                   // number of elements
a.dim1();          // number of elements in a row
a.dim2();          // number of elements in a column
int* p = a.data();          // extract data as a pointer to a C-style array
a(i,j);                     // (i,j)th element (Fortran style), but range checked
a[i];                       // ith row (C-style), but range checked
a[i][j];                    // (i,j)th element C-style

Matrix<int> 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)
a.swap_rows(7,9);           // swap rows a[7] ⇔ a[9]
```

3D Matrix (very like 1D and 2D)

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

Matrix<int> a2 = a; // copy initialization
a = a2;                     // copy assignment
a *= 7;                     // scaling (and +=, -=, /=, etc.)
a.apply(f);                 // a(i,j,k)=f(a(i,j)) for each element a(i,j,k)
a.apply(f,7);               // a(i,j,k)=f(a(i,j),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]
```

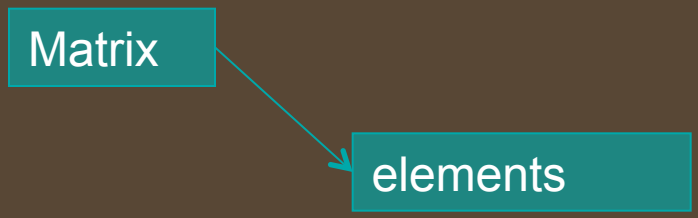
Using Matrix

- See book
  - Matrix I/O
    - §24.5.4; it's what you think it is
  - Solving linear equations example
    - §24.6; it's just like your algebra textbook says it is

Implementing Matrix

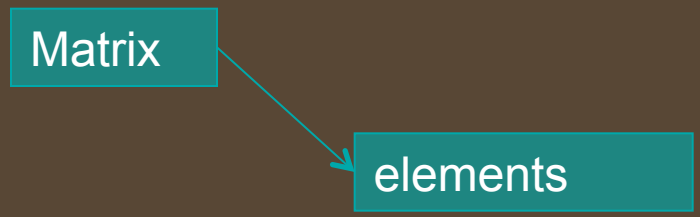
- A Matrix is a handle
  - Controlling access to a sequence of elements
- Let's examine a simplified and improved variant of the idea
  - That is, ***not*** exactly what's provided as **Matrix.h**

```
template<class T, int N> class Matrix {
public:
    // interface, as described
protected:
    T* elem;                // elements stored as a linear sequence
    array<Index,N> extents;     // number of elements in each dimension
};
```



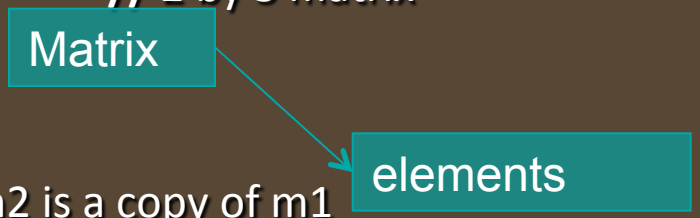
Implementing Matrix

- Resource management:
  - A Matrix must manage the lifetimes of its elements
  - Constructors allocate memory for elements and initialize elements
  - The destructor destroys elements and deallocates memory for elements
  - Assignment copies elements



Implementing Matrix

- We need constructors:
  - Constructor for default elements
    - **Matrix<T,N>::Matrix(Index,Index);**
    - **Matrix<int,2> m0(2,3);** // 2 by 3 matrix each element initialized to 0
  - Initializer-list constructor (C++11)
    - **Matrix<T,N>::Matrix(std::initializer_list<T>);**
    - **Matrix<int,2> m1 = {{1,2,3},{4,5,6}};** // 2 by 3 matrix
  - Copy constructor
    - **Matrix<T,N>::Matrix(const Matrix&);**
    - **Matrix<int,2> m2 = m1;** // m2 is a copy of m1
  - Move constructors (C++11)
    - **Matrix<T,N>::Matrix(Matrix&&);**
    - **return m1;** // move m1 to somewhere else

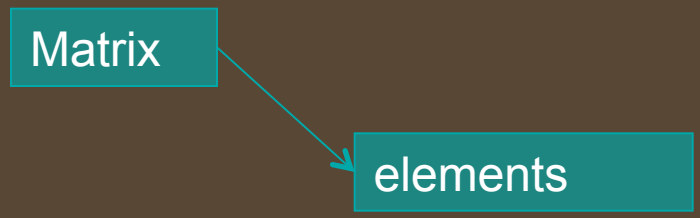


How to move a Matrix

- Idea #1:
  - Return a pointer to a **new**'d object

```
Matrix* operator+(const Matrix&, const Matrix&);
Matrix& res = *(a+b);          // ugly! (unacceptable)
```

  - Who does the **delete**?
    - there is no good general answer

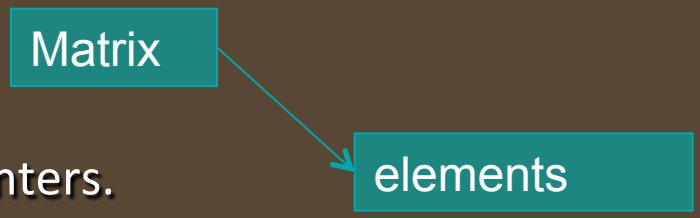


How to move a Matrix

- Idea #2
  - Return a reference to a **new**'d object

```
Matrix& operator+(const Matrix&, const Matrix&);
Matrix res = a+b;        // looks right, but …
```

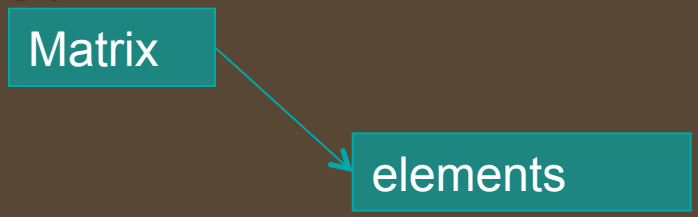
  - Who does the **delete**?
    - What **delete**? I don't see any pointers.
    - there is no good general answer



How to move a Matrix

- Idea #3
  - Pass an reference to a result object

```
void operator+(const Matrix&, const Matrix&, Matrix& result);
Matrix res = a+b;                 // Oops, doesn't work for operators
Matrix res2;
operator+(a,b,res2);              // Ugly!
```



  - We are regressing towards assembly code

How to move a resource

- Idea #4
  - Return a **Matrix**

```
Matrix operator+(const Matrix&, const Matrix&);
Matrix res = a+b;
```

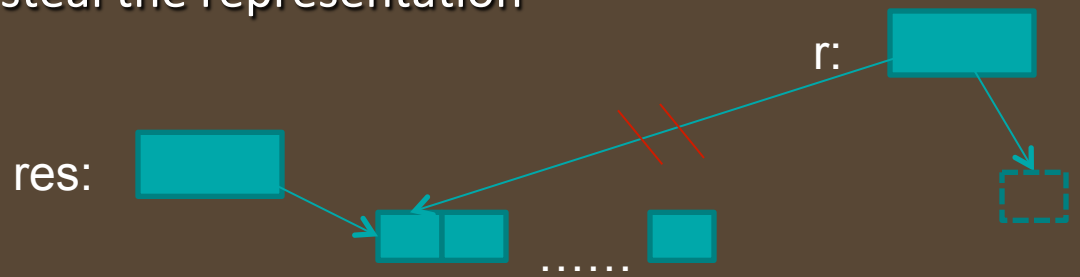
  - Copy?
    - expensive
  - Use some pre-allocated "result stack" of **Matrix**es
    - A brittle hack
  - Move the **Matrix** out
    - don't copy; "steal the representation"
    - Directly supported in C++11 through move constructors

Move semantics

- Return a **Matrix**

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
        Matrix r;
        // copy a[i]+b[i] into r[i] for each i
        return r;
}
Matrix res = a+b;
```

- Define move a constructor for **Matrix**
  - don't copy; "steal the representation"

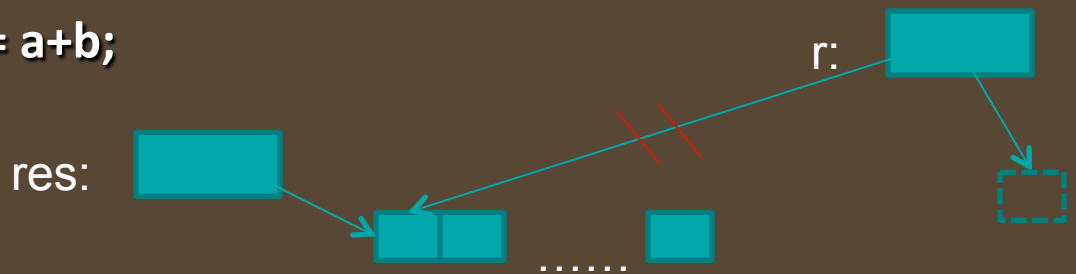


Move semantics

- Direct support in C++11: Move constructor

```
class Matrix {
        // …
        Matrix(Matrix&& a)          // move constructor
        {
                elem = a.elem;      // *this gets a's elements
                a.elem = nullptr;   // a becomes the empty Matrix
                extents = a.extents;        // copy extents
        }
};

Matrix res = a+b;
```



Random numbers

- A "random number" is a number from a sequence that matches a distribution, but where it is hard to predict the next number in the sequence
  - Avoid C-style random numbers from **<cstdlib>**
  - Use **<random>** (C++11)

```
class Rand_int {
public:
        Rand_int(int low, int high) :dist{low,high} { }
        int operator()() { return dist(re); }       // draw an int
private:
        default_random_engine re;                   // generates random numbers
        uniform_int_distribution<> dist;            // makes the distribution uniform
};
```

Random numbers

- Make a histogram

```
int main()
{
        constexpr int max = 9;
        Rand_int rnd {0,max};          // make a uniform random number generator

        vector<int> histogram(max+1) ;          // make a vector of appropriate size


        for (int i=0; i!=200; ++i)                      // fill the histogram
                ++histogram[rnd()];

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

Random numbers

- We get

```
0    **********
1    **********
2    **********
3    **********
4    **********
5    **********
6    **********
7    **********
8    **********
9    **********
```

Complex

- Standard library complex types from **<complex>**

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

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

    // operators:  =  +=  -=  *=  /=
};

// operators:  +  -  /  *  ==  !=

// whatever standard mathematical functions that apply to complex:
// pow(), abs(), sqrt(), cos(), log(), etc. and also norm()  (square of abs())
```

Complex

```
// use complex<T> exactly like a built-in type, such as double
// just remember that not all operations are defined for a complex (e.g. <)

typedef complex<double> dcmplx; // sometimes complex<double> gets verbose

void f( dcmplx z, vector< complex<double> >& vc)
// C++11 allows vector<complex<double>> with no space between > >
{;
    dcmplx z2 = pow(z,2);
    dcmplx z3 = z2*9+vc[3];
    dcmplx  sum = accumulate(vc.begin(), vc.end(), dcmplx());
}
```

Numeric limits

- Each C++ implementation specifies properties of the built-in types
  - used to check against limits, set sentinels, etc.
- From **<limits>**
  - for each type
    - **min()** // smallest value, e.g., **numeric_limits<int>::min()**
    - **max()** // largest value
    - …
  - For floating-point types
    - Lots (look it up if you ever need it)
    - E.g. **numeric_limits<float>::max_exponent10**
- From **<climits>** and **<cfloat>**
  - **INT_MAX** // largest **int** value
  - **DBL_MIN** // smallest positive **double** value

Numeric limits

- They are important to low-level tool builders
- If you think you need them, you are probably too close to hardware, but there are a few other uses. For example,

```
void f(const vector<int>& vc)
{
    // pedestrian (and has a bug):
    int smallest1 = v[0];
    for (int i = 1; i < vc.size(); ++i) if (v[i] < smallest1) smallest1 = v[i];

    // better:
    int smallest2 = numeric_limits<int>::max();
    for (int i = 0; i < vc.size(); ++i) if (v[i] < smallest2) smallest2 = v[i];

    // or use standard library:
    vector<int>::iterator p = min_element(vc.begin() ,vc.end());
    // and check for p==vc.end()
}
```