# Numerics

2016/04/12

# Outline

# Floating Point Numbers





- The computer has to save numbers such as $1/3 = 0.3333\ldots$ with a finite amount of memory.
- Solution: floating point numbers:

$$(-1)^s \times (a_1.a_2a_3\ldots a_t) \times b^e = (-1)^s \times m \times b^e$$
$$1 \leq a_1 < b$$

- Terms

  $m$ mantissa
  $t$ number of digits
  $b$ basis
  $e$ exponent

- Example: $3.3333 \times 10^{-1}$
- Standard type in C++: `double`

# General implications





- Only a finite range of numbers can be handled by a data type
- Errors results when the range is left

  Overflow number is too high for a type
  Underflow number is too small for a type
- Numbers can only be stored approximately
- Rounding errors result

# Implications for ==

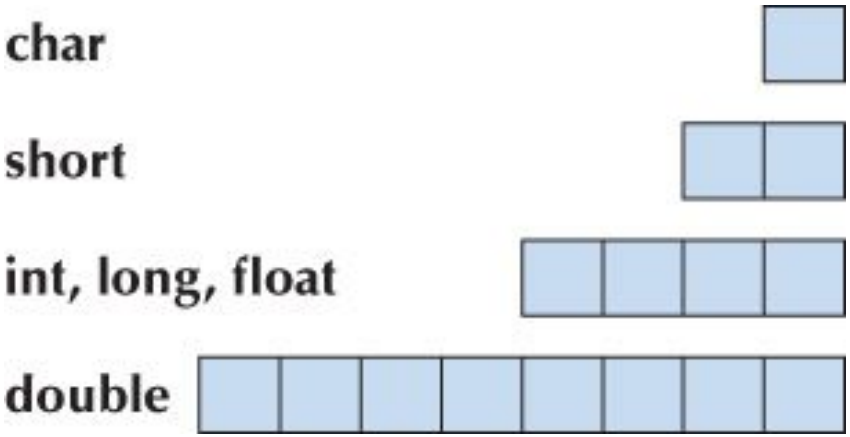
- Given a basis $b$ some numbers can be stored precisely, some not
  - $1/2$ with $b = 10$ and 3 digits: $5.00 \times 10^{-1}$
  - $1/3$ with $b = 10$ and 3 digits: $3.33 \times 10^{-1}$
- So ....

```
#include<iostream>

using namespace std;

int main()
{
  bool b1 = 0.15*3 == 0.45;    // false
  bool b2 = 0.16*2 == 0.32;    // true

  cout << b1 << endl << b2 << endl;

  return 0;
}
```

# Type Sizes in Bytes








- When a `float` is assigned to an `int` digits get **truncated**: $3.14 \rightarrow 3$
- When an `int` is assigned to a `float` you might loose **precision** when your `int` is too high
  $2100000009 \rightarrow 2100000000$

# C++ Language Facilities





- Header files: `<limits>`, `<climits>`, `<limits.h>`, `<float.h>`
- `<limits>` gives `numeric_limits<T>` for built-in types with member functions such as `min()`, `max()`, `lowest()`, `epsilon()` and many more ...

# C++ Language Facilities





```
#include<iostream>
#include<limits>

using namespace std;

int main(){

  cout << "DOUBLE_MAX:\t" << numeric_limits<double>::max() << endl
       << "DOUBLE_MIN:\t" << numeric_limits<double>::min() << endl
       << "DOUBLE_LOWEST:\t" << numeric_limits<double>::lowest() << endl;
  // check std::numeric_limits for more :)
  return 0;
}
```

# The Matrix Library

## Why do we need one?





- In principle we could use arrays.
- For example

```
double my_matrix[3][4];
// declares a 3 x 4 matrix
```

- This is a bad idea.

# The Matrix Library

## Access, Indexing



```
#include<iostream>
#include"Matrix.h"

using namespace Numeric_lib;

int main(){
  int n{2};
  int m{4};

  Matrix<double,2> mymat(n, m);   // 2 - dim. matrix

  mymat(1,2) = 2.0;               // 2,3 - element!
  mymat(0,1) = 4.2;               // 1,1 - element!
  double elem12 = mymat[0][1];
  std::cout << elem12 << std::endl;

  return 0;
}
```

# The Matrix Library

## Dimensions



- A matrix knows its dimensions.

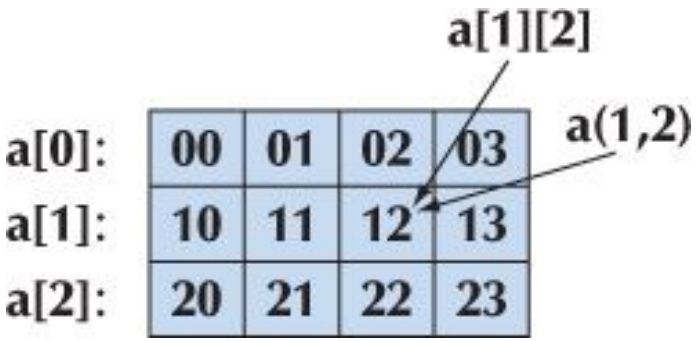
```
#include<iostream>
#include"Matrix.h"

using namespace Numeric_lib;

int main(){
  int n{2}, m{4};

  Matrix<double,2> mymat(n, m);

  std::cout << "Rows: " << mymat.dim1() << std::endl
            << "Columns: " << mymat.dim2() << std::endl
            << "No Elements: " << mymat.size() << std::endl;

  return 0;
}
```

# The Matrix Library

## Dimensions II



- The dimensions are part of the matrix type. So you cannot write functions that take a general matrix. You would have to write a template for that.
- Matrix is stored in memory in row-first fashion:




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

# The Matrix Library

## Slicing



- Works with one-dimensional and two - dimensional matrices

```
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)
```

# The Matrix Library

## Some common matrix functions





```
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)


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

# Matrix I/O





```
#include<iostream>
#include"Matrix.h"
#include"MatrixIO.h"

using namespace Numeric_lib;

int main(){

  double val[] = {1.2, 3.4, 5.6, 7.8};
  Matrix<double> a(val);
  std::cout << a << std::endl;

  Matrix<double,2> b(2,2);
  std::cout << "Type␣in␣a␣matrix␣of␣the␣form" << std::endl
            << "{" << std::endl
            << "{a␣b}" << std::endl
            << "{c␣d}" << std::endl
            << "}" << std::endl;

  std::cin >> b;
  std::cout << b << std::endl;
```

# Gaussian Elimination



- Problem: Solve the linear system

$$Ax = b$$

for $x$, where $A$ is quadratic $(n \times n)$

- Solution: transform both sides of the equation such that $A^*$ in

$$A^*x = b^*$$

is upper-triangular.

- Depending on the entries of $A^*$, the equation has zero, one or infinitely many solutions.

# Gaussian Elimination

## Solving the Upper-Triangular Form - Details



- The system looks should like this at the end of the elimination

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

- Solve for $x[n] = b[n]/a(n,n)$, eliminate that row and solve for $x[n-1]$ and so on.
- Works if all diagonal elements are non-zero, otherwise system has none or infinitely many solutions.

# Random Numbers





- The sl header `<random>` has it.
- Structure is

  **Engine** to generate uniformly distr. random integer(!) numbers

  **Distributions** map random numbers to distributions

# Mathematical Functions & Complex Numbers





- Mathematical functions can be found in <cmath>. They have the usual names, that is, `cos`, `sin`, `floor` etc.
- Complex numbers are in <complex>.
- For some weird reason, complex numbers are implemented as a template, so you have to initialize like

```
#include<iostream>
#include<complex>

using namespace std;

int main(){

  complex<double> mynumber(2,1);
  cout << mynumber << endl
       << "Real part: " << real(mynumber) << endl
       << "Imag. part: " << imag(mynumber) << endl;

  return 0;
}
```

# The End





Merci for listening.