ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Εισαγωγή στον Προγραμματισμό
Introduction to Programming

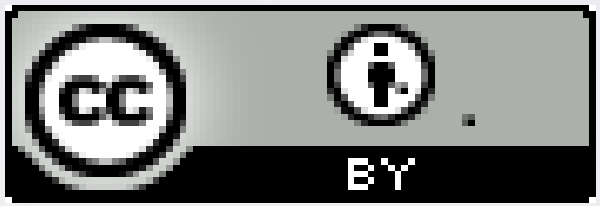
Διάλεξη 21: Επεξεργασία Κειμένου και Αριθμών

Γ. Παπαγιαννάκης

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται στην άδεια χρήσης **Creative Commons** και ειδικότερα

Αναφορά Δημιουργού 3.0 - Μη εισαγόμενο Ελλάδα (Attribution 3.0– Unported GR)



- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

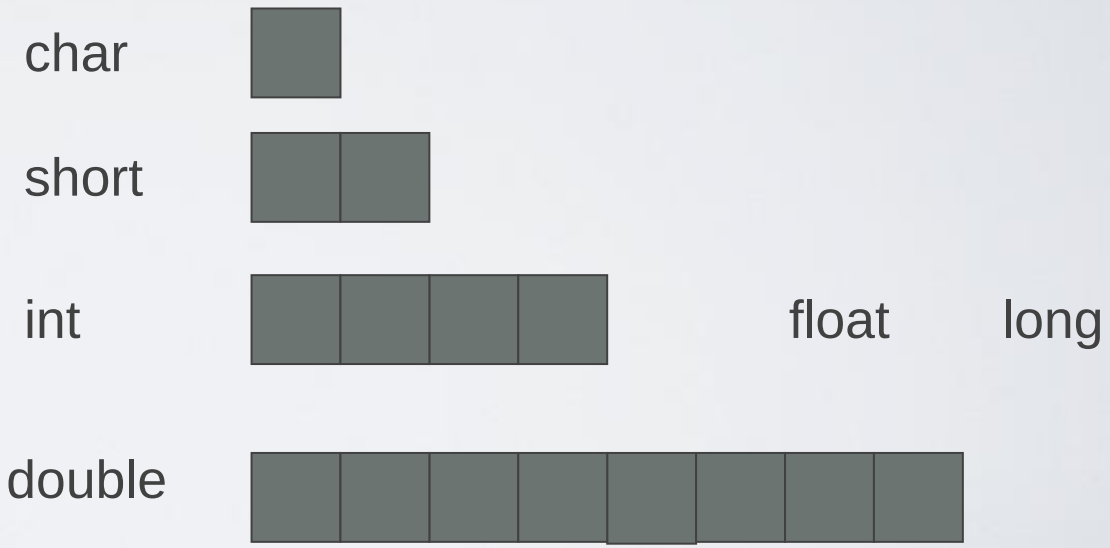
HY-150 Προγραμματισμός
CS-150 Programming

Lecture 21:
Text and Numerical Processing

G. Papagiannakis

Overview

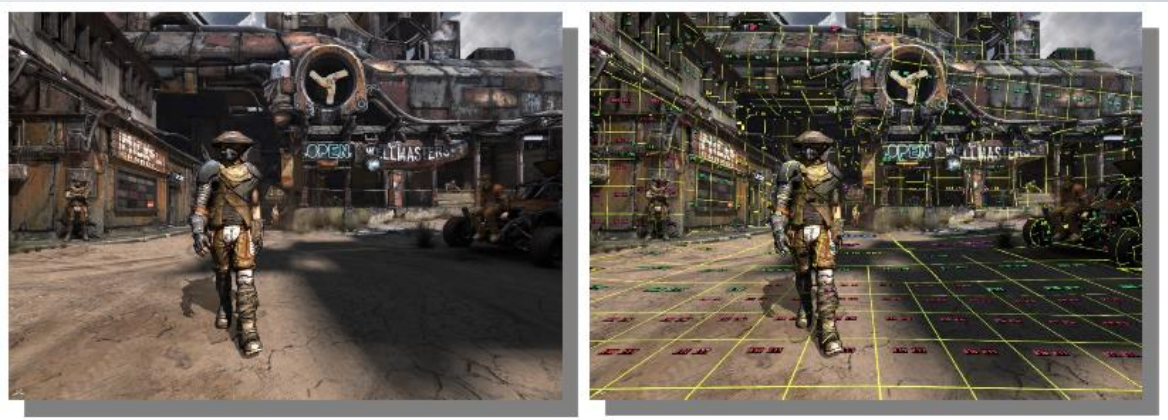
- Strings
- I/O
- Precision, overflow, sizes, errors
- Matrices
- Random numbers
- Complex numbers
- examples

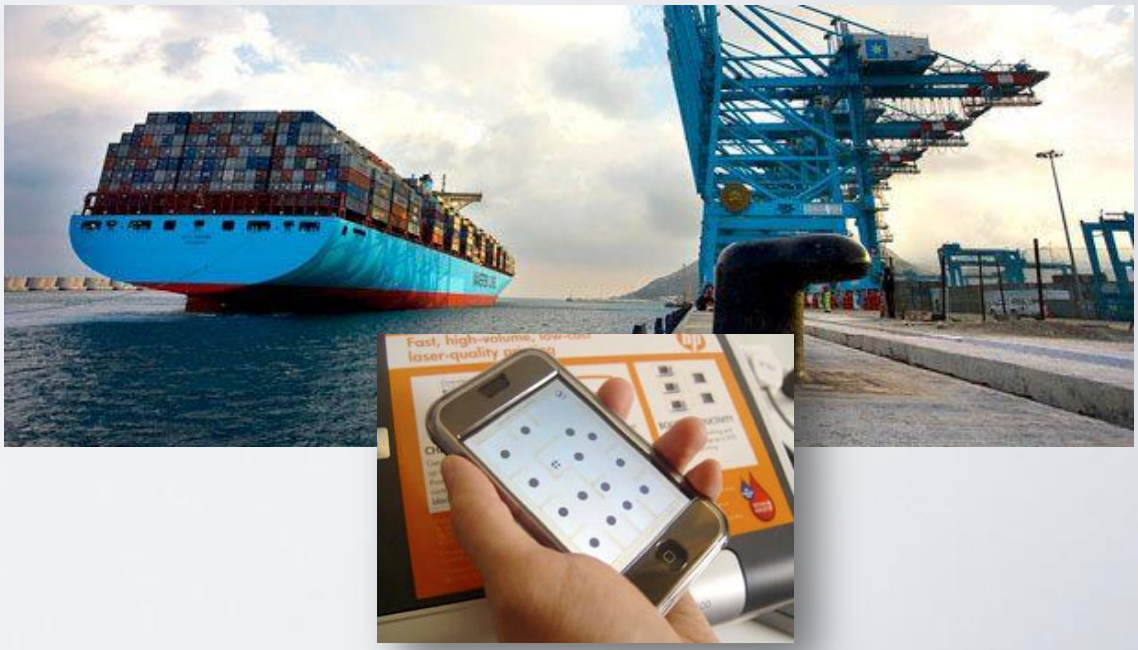


Now you know the basics

- Really! Congratulations!

- Don’t get stuck with a sterile focus on programming language features
- What matters are programs, applications, what good can you do with programming
  - Text processing
  - Numeric processing
  - Embedded systems programming
  - Banking
  - Medical applications
  - Scientific visualization
  - Interaction & gaming
  - Route planning
  - Physical design
  - Mobile computing





Text processing

- “all we know can be represented as text”
  - And often is
- Books, articles
- Transaction logs (email, phone, bank, sales, …)
- Web pages (even the layout instructions)
- Tables of figures (numbers)
- Mail
- Programs
- Measurements
- Historical data
- Medical records
- …

String overview

- Strings
  - **std::string**
    - **<string>**
    - **s.size()**
    - **s1==s2**
  - C-style string (zero-terminated array of char)
    - **<cstring>** or **<string.h>**
    - **strlen(s)**
    - **strcmp(s1,s2)==0**
  - **std::basic_string<Ch>**, e.g. unicode strings
    - **typedef std::basic_string<char> string;**
  - Proprietary string classes

String conversion

- Simple to_string

```
template<class T> string to_string(const T& t)
{
    ostringstream os;
    os << t;
    return os.str();
}
```

- For example:

```
string s1 = to_string(12.333);
string s2 = to_string(1+5*6-99/7);
```

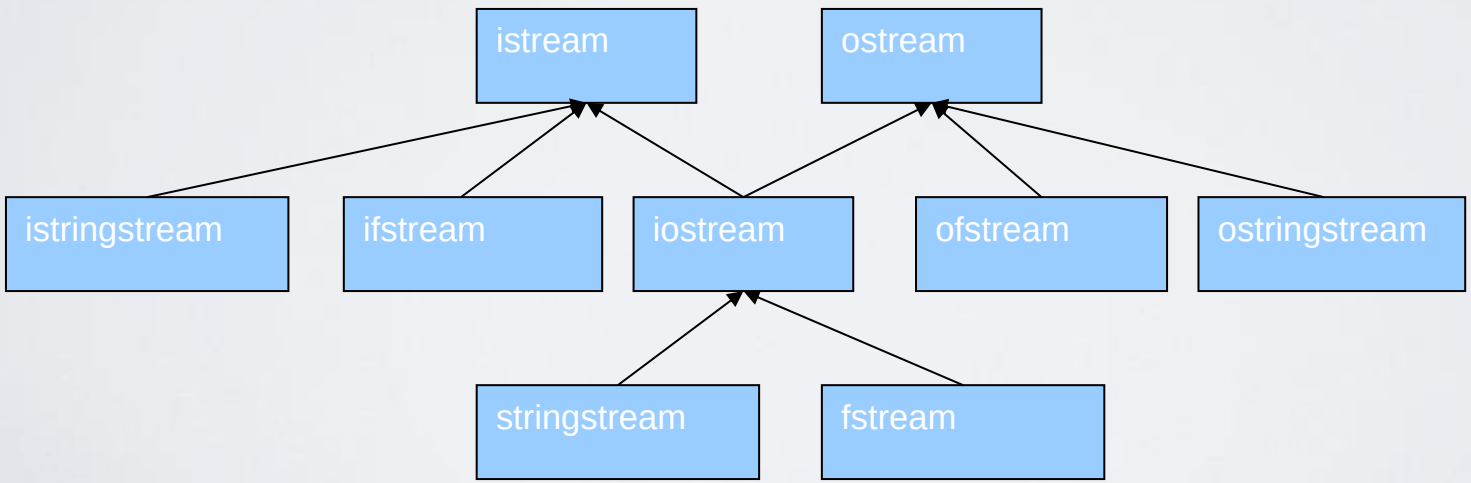
Selected string operations

- <string>

Selected string operations	
s1 = s2	Assign **s2** to **s1**; **s2** can be a **string** or a C-style **string**.
s += x	Add **x** at end; **x** can be a character, a **string**, or a C-style string.
s[i]	Subscripting.
s1+s2	Concatenation; the characters in the resulting **string** will be a copy of those from **s1** followed by a copy of those from **s2**.
s1==s2	Comparison of **string** values; **s1** or **s2**, but not both, can be a C-style string. Also **!=** .
s1<s2	Lexicographical comparison of **string** values; **s1** or **s2**, but not both, can be a C-style string. Also **<=**, **>**, and **>=**.
s.size()	Number of characters in **s**.
s.length()	Number of characters in **s**.
s.c_str()	C-style version of characters in **s**.
s.begin()	Iterator to first character.
s.end()	Iterator to one beyond the end of **s**.
s.insert(pos,x)	Insert **x** before **s[pos]**; **x** can be a character, a **string**, or a C-style string. **s** expands to make room for the characters from **x**.
s.append(pos,x)	Insert **x** after **s[pos]**; **x** can be a character, a **string**, or a C-style string. **s** expands to make room for the characters from **x**.
s.erase(pos)	Remove the character in **s[pos]**. **s**'s size decreases by 1.
pos = s.find(x)	Find **x** in **s**; **x** can be a character, a **string**, or a C-style string; **pos** is the index of the first character found, or **npos** (a position off the end of **s**).
in>>s	Read a whitespace-separated word into **s** from **in**.
getline(in,s)	Read a line into **s** from **in**.
out<<s	Write from **s** to **out**.

I/O overview

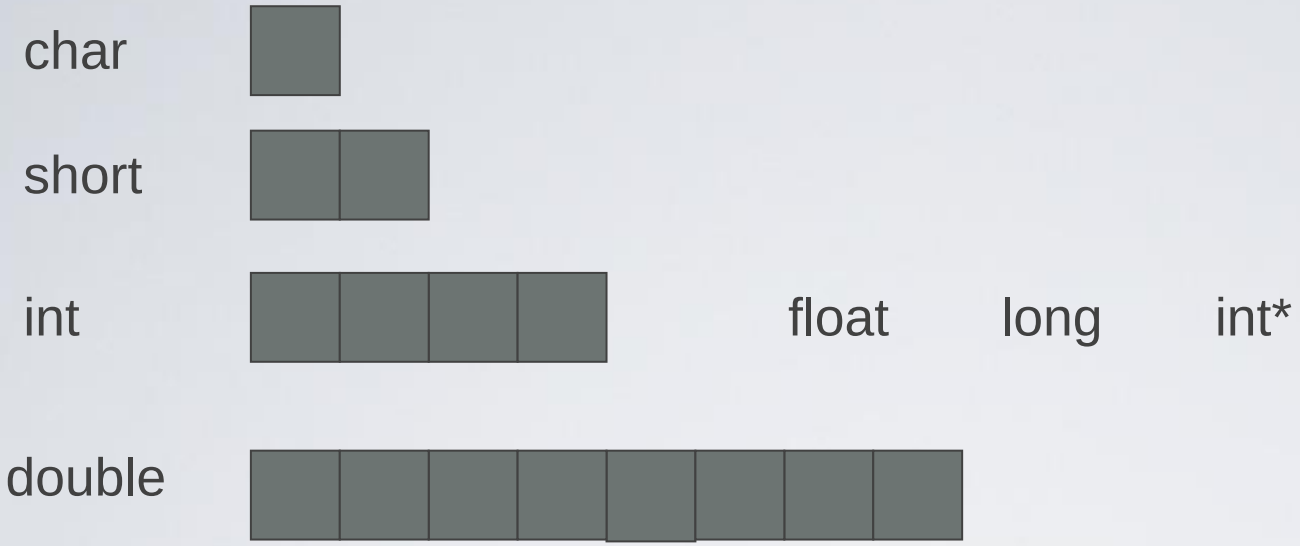
Stream I/O	
in >> x	Read from **in** into **x** according to **x**'s format
out << x	Write **x** to **out** according to **x**'s format
in.get(c)	Read a character from **in** into **c**
getline(in,s)	Read a line from **in** into the string **s**



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

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
   }
   // …
}
```

Math function errors

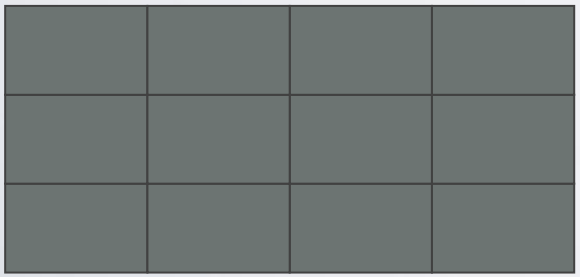
- If a standard mathematical function can't do its job, it sets **errno from <cerrno>**, for example

```
void f(double negative, double very_large)
   // primitive (1974 vintage, pre-C++) but ISO standard error handling
{
   errno = 0;              // no current errors
   errno=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";
   pow(very_large,2);         // not a good idea
   if (errno==ERANGE)         // range error
         cerr << "pow(" << very_large << ",2) too large for a double";
}
```

Matrices

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

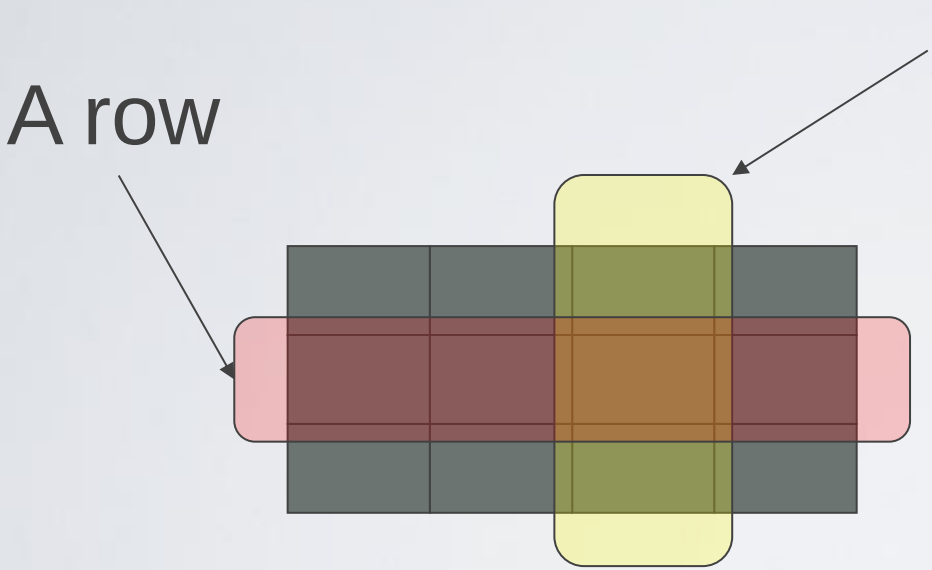
A vector (e.g. **Matrix<int> v(4)**), also called a 1 dimensional Matrix, or even a 1*N matrix



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++PL, Appendix C, pp 836-840

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 course site 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[2]**)
  - 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")

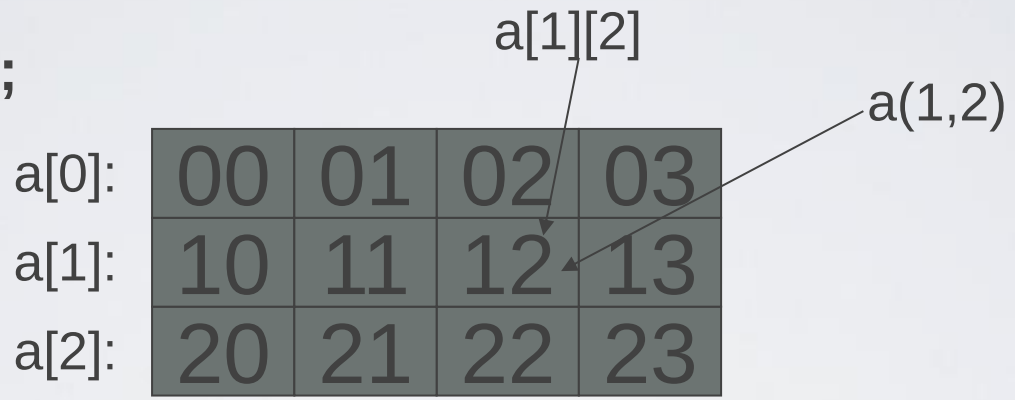
A Matrix library

```
// 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:

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 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 asssignment
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 asssignment
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 asssignment
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

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
  - Uniformly distributed integers **<cstdlib>**
    - **int rand()** *// a value in [0:RAND_MAX]*
    - **RAND_MAX** *// the largest possible value for rand()*
    - **void srand(unsigned);** *// seed the random number generator*
  - Use
    - **int rand_int(int max) { return()%max; }**
    - **int rand_int(int min, int max) {return min+rand_int(max-min); }**
  - If that’s not good enough (not “random enough”) or you need a nonuniform distribution, use a professional library
    - E.g. **boost::random** (also C++0x)

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()) { }

    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<float> >& vc)
{;
   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
    - **max()** // largest value
    - …
  - For floating-point types
    - Lots (look it up if you ever need it)
    - E.g. **max_exponent10()**
- From **<limits.h>** and **<float.h>**
  - **INT_MAX** // largest **int** value
  - **DBL_MIN** // smallest **double** value

Numeric limits

- They are important to low-level tool builders
- If you think 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()
}
```

A great link

- http://www-history.mcs.st-andrews.ac.uk/index.html
- A great link for anyone who likes math
  - Or simply needs to use math
  - Famous mathematicians
    - Biography, accomplishments, plus curio,
      for example, who is the only major mathematician to win an Olympic medal?
  - Famous curves
  - Famous problems
  - Mathematical topics
    - Algebra
    - Analysis
    - Numbers and number theory
    - Geometry and topology
    - Mathematical physics
    - Mathematical astronomy
  - The history of mathematics
  - …

Application domains

- Text and numerical processing is just one domain among many
  - Or even several domains (depending how you count)
  - Browsers, Word, Acrobat, Visual Studio, …
- Image processing
- Sound processing
- Computer graphics
- Data bases
  - Medical
  - Scientific
  - Commercial
  - …
- Biotech
- Financial
- …

Acknowledgements

Bjarne Stroustrup

Programming -- Principles and Practice Using C++

http://www.stroustrup.com/Programming/

Thank you!


BY


Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης