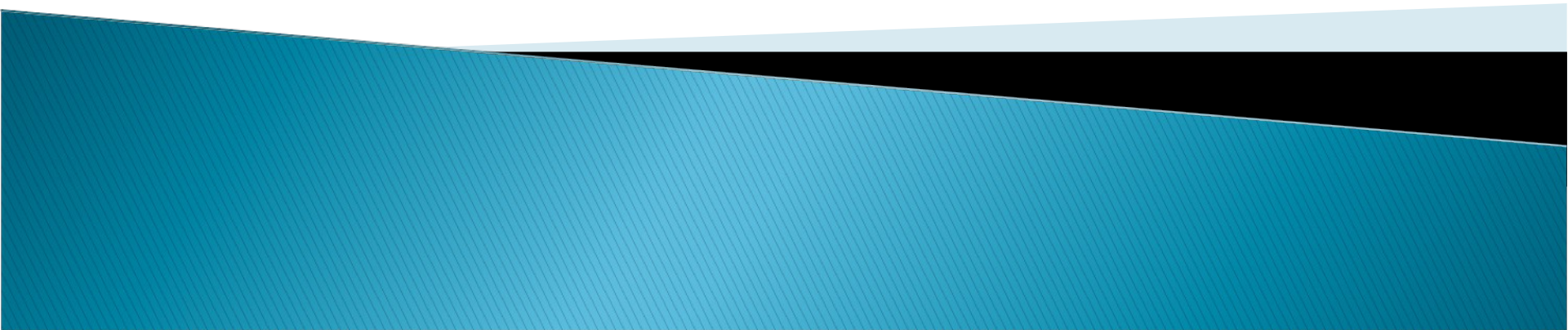
Math 6370 Lecture 2: C++ Introduction for Matlab Programmers

Daniel R. Reynolds

Spring 2013
reynolds@smu.edu



Example C++ Program

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

// Example C++ routine
int main(int argc, char* argv[]) {

  // declarations
  int i, n;
  double *a, *b, sum, rtime;
  clock_t stime, ftime;

  // ensure that an argument was passed in
  if (argc < 2) {
    printf("Error: requires 1 argument\n");
    return 1;
  }

  // set n as the input arg, ensure positive
  n = atoi(argv[1]);
  if (n < 1) {
    printf("Error: arg must be >0\n");
    return 1;
  }

  // allocate the vectors
  a = new double[n];
  b = new double[n];

  // initialize the vector values
  for (i=0; i<n; i++) {
     a[i] = (0.001 * (i + 1.0)) / n;
     b[i] = (0.001 * (n - i - 1.0)) / n;
  }

  // compute dot-product
  stime = clock();
  sum = 0.0;
  for (i=0; i<n; i++)   sum += a[i]*b[i];
  ftime = clock();
  rtime = ((double) (ftime - stime))
          /CLOCKS_PER_SEC;

  // output computed value and runtime
  printf(" vector length = %i\n",n);
  printf("   dot-product = %.16e\n",sum);
  printf("      run time = %.2e\n",rtime);

  // delete vectors
  delete[] a;
  delete[] b;

  return 0;
} // end main
```

C++ Character Set, Variables

- C++ recognizes the alphanumeric characters:
  - A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  - a b c d e f g h i j k l m n o p q r s t u v w x y z _
  - 0 1 2 3 4 5 6 7 8 9
  - + - * / = ! @ # $ % & ' “ < > ? / \ : ; ,

- Names of variables, functions, classes, etc., must not begin with a digit. These may include letters, numbers and _, but cannot include spaces or other punctuation marks, and are case-dependent (i.e. Name != nAmE).
  - Good names: max_norm, TotalEnergy, L1Norm
  - Bad names: 2norm, dot product, Hawai'i, x-velocity

Reserved Keywords

You should avoid names that match reserved keywords:

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

Some words are also reserved under certain circumstances, and should be avoided to maximize program portability:

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

Operators, Special Characters

Arithmetic Operators:

- +
- -
- *
- /
- %

Relational Operators:

- <, >
- <=, >=
- ==
- != (~= in Matlab)

Logical Operators:

- && (& in Matlab)
- || (| in Matlab)
- ! (~ in Matlab)

Increment/Decrement Operators (assume a=6):

- --:

```
x = a--     (x=6, a=5)
x = --a     (x=5, a=5)
```

- ++:

```
x = a++     (x=6, a=7)
x = ++a     (x=7, a=7)
```

SMU

Operators, Special Characters

Conditional operator:

- condition ? statement : statement

```
mx = (a > b) ? a : b;
```

Shorthand operators:

- `+=`
- `-=`
- `*=`
- `/=`

Allocation/deallocation operators:

- Allocation: `new`
- Deallocation: `delete`

Special characters:

- Single line comment: //
- Multi-line comment: /* */
- Struct/Class element: .
- Ptr to Struct/Class element: ->
- Dereference operator: *
- Reference operator: &
- Code block: { }
- Scope resolution operator: ::
- List separator in variable declaration: ,

Notation

In the rest of the lectures for this course, we will strive to use the following notation:

- *Italicized* words will denote items that you may name yourself
- [bracketed] words will denote optional items
- ... will denote areas for commands (or Matlab line continuation)
- `monospaced` font will be used to denote code segments
- green words in code blocks will denote comments

Every effort will be made to compare/contrast C++ commands, syntax, and usage with Matlab. If at any point you are unclear about something, please stop me to ask a question.

In working with codes, I suggest turning on syntax highlighting in emacs ('Options' menu), which will colorize keywords to make coding easier.

Program Structure

In general, C/C++ programs are structured similarly to Matlab programs, except that since C/C++ are compiled (Matlab is interpreted), there is no command prompt, so all code must be included in a main program and functions (not typed at the prompt).

As a result, the code may be written using any text editor of your choosing (I like emacs), and you must tell the compiler that the file contains the main program with the “main” keyword.

Matlab – Script file:

```
…
```

C/C++ – Main routine:

```
int main(int argc, char* argv[])
{
    …
}
```

Functions without return variables

Routines that take input arguments but do not return anything (e.g. that print things to the screen or to an output file), are almost identical between Matlab and C/C++, except that in C/C++ the function must be enclosed in { }, a single file can contain *many* C/C++ functions, and the file name need not match the function name.

Matlab:

```
function fname(arg1, arg2)

   ...

return
```

C/C++:

```
void fname(type arg1, type arg2) {

   ...

   return;   // optional
}
```

Functions with return variables

Functions with only a single output are almost identical, except that the C/C++ function output *data type* (we'll discuss this soon) must be specified when declaring the function.

Matlab:

```
function y = fname(arg1, arg2)

    …
    y = …

return
```

C/C++:

```
int fname(type arg1, type arg2)
{
    int y;
    …
    y = …
    return y;
}
```

SMU

Difference: Call By Reference/Value

Functions with multiple outputs must be handled differently, since C/C++ do not allow you to specify multiple output types for a function.

The solution highlights a significant difference between Matlab and C/C++. While Matlab makes copies of input arguments to use locally in the function, C/C++ can operate on the input arguments directly (more on this later).

Matlab: "call by value"

Multiple-output function:

```
function [x,y] = fname(a)

  …
  x = …
  y = …

  return
```

C/C++: "call by reference"

Multiple-output function:

```
void fname(type a, type *x,
           type *y)
{
   …
   *x = …
   *y = …
   …
   return;
}
```

SMU

Pointers: the key to “call by reference”

- The address that locates a variable within memory is what we call a *reference* to that variable.
  - This can be obtained by preceding the variable with a (&), which can be literally translated as "address of".
  - For example:

    ```
    a = &b;
    ```

    assigns `a` to the memory address that stores `b`.

- A variable which stores a reference to another variable is called a *pointer,* which is said to "point to" the variable whose reference they store.
  - These can directly access the value stored in the variable which it points to.
  - To do this, we precede the pointer's identifier with a (*), which can be literally translated to "value pointed by".
  - For example:

    ```
    c = *a;
    ```

    assigns `c` to the value pointed to by `a`.

Difference: Data Types

A *data type* is the computer's format for representing a number, object, or memory reference using a given number of bits.

Matlab:

- All variables default to a double precision matrix, unless otherwise specified.
- Variables may be declared (and redeclared) anywhere in a program or function.

C/C++:

- We must specify the data type (and size) for every variable.
- Variables may still be declared anywhere within a program or function, but it is "good style" to declare things at the top of each routine. Variables may never be redeclared within a given program scope.
- If a variable is sent by reference to a function, it *may be changed by that function* prior to returning to the calling routine.

Data Types: Integers

C/C++ have three default kinds of integers:

- `short int` – uses 16 bits (2 bytes) of memory:
  - signed: IEEE min = -32,768; max = 32,768
  - unsigned: IEEE min = 0; max = 65535
- `int` – uses 32 bits (4 bytes) of memory:
  - signed: IEEE min = -2,147,483,647; max = 2,147,483,647
  - unsigned: IEEE min = 0; max = 4,294,967,295
- `long int` – uses 64 bits (8 bytes) of memory:
  - signed: IEEE min = -9,223,372,036,854,775,807;
    max = 9,223,372,036,854,775,807
  - unsigned: IEEE min = 0; max = 18,446,744,073,709,551,614

Be careful with integer arithmetic: In order to achieve an integer result, non-integer outputs must be rounded (the default is to floor toward 0):

```
1/2 = 0
```

Data Types: Floating-Point

C/C++ have three default kinds of floating-point numbers:

- `float` – uses 32 bits (4 bytes) of memory:
  - IEEE max ~$3.4*10^{38}$
  - Precision: ~7 digits
- `double` – uses 64 bits (8 bytes) of memory:
  - IEEE max ~$1.8*10^{308}$
  - Precision: ~16 digits
- `long double` – uses 128 bits (16 bytes) of memory:
  - IEEE max ~$1.2*10^{4932}$
  - Precision: ~32 digits

C/C++ have no built-in "complex" type, though the C++ standard library does provide a "complex" class, allowing each of the floating-point types above.

Data Types: Other

- Char – character or small integer, uses 1 byte (8 bits)
  - Letters, e.g.: A, b, _
  - Unsigned integers from -128 through 127
- Bool – boolean value (`true`, `false`), uses 1 byte (8 bits)
- Any variable may be declared as a pointer to the relevant type by adding a (*), e.g.

```
long double *value;
```

Custom Data Types: Structures

C (and by default C++) allows you to package variables together into structures, e.g.

```
struct vector {
   float x;
   float y;
};


int main() {
  vector u;
  u.x = 5.0;
  u.y = 1.0;

  vector *v = new vector;
  v->x = 2.0;
  (*v).y = 3.0;

  return 0;
}
```

- Defines a new type, `vector`, containing two `float` values, `x` and `y`.
- Creates a vector `u`, with components (5,1).
- Allocates memory for a vector (using `new`), and sets a pointer to that memory location, `v`. Then sets the components (2,3) into the vector that `v` points to.

Custom Data Types: Classes

- C++ allows you to define structures that can have functions associated with themselves.
- An *object* is just an instantiation of a class.

```
class rect {
   float x, y;
  public:
   void setrect(float,float);
   float area() {return x*y;};
};

void rect::setrect(float a, float b){
   x = a; y = b;
}

int main() {
  rect R;
  R.setrect(2.0, 3.0);
  float A = R.area();
  return 0;
}
```

- Defines a new class, `rect`, containing two `float` values, `x` and `y`, and two routines, `setrect()` and `area()`.
- Creates a rect R, with components (2,3).
- R then computes its area, and the result is stored in the float A.

Data Types: Declaring Constants

Be careful with constants, since the compiler does not know your intended type:

```
1/3  !=  1.0/3.0  !=  1.0f/3.0f
```

```
// integer constants
A = 75;          // int (the default)
B = 75u;         // unsigned int
C = 75l;         // long int
D = 75ul;        // unsigned long int

// floating-point constants
E = 3.14159f;  // float
F = 3.14159;   // double (the default)
G = 3.14159L;  // long double
```

Arrays

- We can declare an array of entries of any intrinsic (or user-defined) type:

```
float a[10];  // declares array a of 10 32-bit numbers
int* b[5];    // declares array b of 5 ptrs to integers
```

- Arrays in C/C++ begin with “0” and are accessed with square brackets:

```
a[0] = 2.0; a[1] = 5.0;  // sets the 1st & 2nd elements of a
```

- Arrays may be initialized with a list of values, indicated with { , , }

```
int b[4] = {2,3,4,5}; // declares an int array, b, of length 4
```

- If the initializer supplies too few entries, the remaining entries are set to 0:

```
int v[6] = {1,2,3,4};       // these two arrays are identical
int w[6] = {1,2,3,4,0,0};
```

Arrays (continued)

- For arrays with size set at runtime, we use a combination of a pointer and the new (again, using any intrinsic or user-defined type):

```
N = ...;                              // declares x as an array of
double* x = new double[N];            // doubles of length N
```

- These must be subsequently freed to avoid memory leaks:

```
delete[] x;                 // frees all memory associated with x
```

- 2D arrays are allowed:

```
int b[2][3];                // b is an array of 2 arrays of 3 integers
int a[2][2] = {{1, 2}, {3, 4}};   // a is a 2x2 array
```

- But 2D arrays at runtime are messy (and don't get me started about 3D):

```
int** a = new int*[2]; // a is an array of 2 integer arrays,
a[0] = new int[M];     // each of which must be allocated
a[1] = new int[M];     // and deleted separately.
```

Array Operations

- Unlike Matlab (and Fortran90), C/C++ do not allow whole-array operations or array slicing, so all array computations must be performed inside loops:

```
int a[5][6];
for (i=0; i<5; i++)        // this is much uglier than doing
    for (j=0; j<6; j++)    // the same thing in Matlab
        a[i][j] = 2.0;     // (we'll talk about for loops soon)
```

- Moreover, calculations over multi-dimensional arrays are typically *much slower* than arrays allocated contiguously in memory:

```
double** A;    // if both A and B are the same size, calculations
double* B;     // with B will most likely be faster, but require you
               // to *think* about flattening the 2D array into 1D
```

- The C++ standard library implements a “vector” class, but this is not the same as a mathematical vector, and calculations with them are *slow*.

Output

To output values to the screen, you may use:

- `printf()` [like Matlab's `fprintf()`]

```
printf("pi = %.16e\n", pi);
```

- `cout` [easier, but with less formatting control; must `#include <iostream>`]

```
std::cout << "pi =" << pi << std::endl;
```

To output values to disk, you may use:

- `fprintf()` [like `printf()`, but the first arg is now a pointer to the output file]

```
FILE *fptr = fopen("results.txt","w"); // open file for writing
fprintf(fptr, "pi = %.16e\n", pi);     // write output to file
fclose(fptr);                          // close output file
```

- `ofstream` [must `#include <iostream>` and `#include <fstream>`]:

```
std::ofstream fptr;                    // declare output stream
fptr.open("results.txt", std::fstream::out);// open write-only file
fptr << "pi =" << pi << "\n";         // write output to stream
fptr.close();                          // close output stream
```

SMU

Input

We can similarly use `scanf()` and `fscanf()` to read values from stdin/file:

```
int n, m;                  // declare variables
printf("Enter n:\n");      // prompt user for input
scanf("%i", &n);           // read value from stdin
FILE *fptr = fopen("inputs.txt","r");   // open file for reading
fscanf(fptr,"m = %i\n", &m);            // read value from file
fclose(fptr);                           // close file
```

Or we can use `cin` and `ifstream` [`#include <iostream>` and `#include <fstream>`]:

```
int n, m;                       // declare variables
std::cout << "Enter n:\n";      // prompt user for input
std::cin >> n;                  // read value from stdin
std::ifstream fptr;             // declare input stream
fptr.open("inputs.txt", std::fstream::in); // open read-only file
fptr >> m;                      // read value from file
fptr.close();                   // close output stream
```

Control Structures

Matlab:

For loop (fwd/bwd):

```
for i=1:n
    …
end
for i=n:-1:1
    …
end
```

While loop:

```
while (condition)
    …
end
```

C/C++:

For loop (fwd/bwd):

```
for (i=1; i<=n; i++) {
    …
}
for (i=n; i>=1; i--) {
    …
}
```

While loop:

```
while (condition) {
    …
}
```

Control Structures (cont)

Matlab:

If statements:

```
if (condition)
   …
elseif (condition)
   …
else
   …
end
```

C/C++:

If statements:

```
if (condition) {
   …
} else if (condition) {
   …
} else {
   …
}
```

SMU

Control Structures (cont)

Matlab:

Switch statements:

```
switch param
case 0
   …
case 1
   …
case {2,3}
   …
otherwise
   …
end
```

C/C++:

Switch statements:

```
switch (param) {
case 0:
   …
   break;
case 1:
   …
   break;
case 2:
case 3:
   …
   break;
default:
   …
}
```

Control Structures (cont)

Matlab:

Continue/break statements:

```
for i=1:n

   if (i<4)
      continue;
   end

   sprintf('i=%i\n',i);

   if (i>10)
      break;
   end

end
```

C/C++:

Continue/break statements:

```
for (i=1; i<=n; i++) {

   if (i<4) {
      continue;
   }

   printf("i=%i\n",i);

   if (i>10) {
      break;
   }

}
```

Control Structures (cont)

Other important control mechanisms [Matlab equivalent]:

- `return`: **exits the function [`return`]**
- `exit`: **terminates the program immediately [`quit`]**
- `goto`: **allows to make an absolute jump to another point in the program (not recommended) [(no Matlab equivalent)]**

Control Structures: Example

```
void foo(double *a) {          // name, args of function
   int i;                      // declare local variables
   for (i=0; i<100; i++) {     // begin loop
      if (a[i] < 0.0) {        // terminate program on negative value
         exit;
      } else if (a[i] == 0.0) {    // skip zero-valued entries
         continue;
      } else if (a[i] > 1.e20) {   // stop loop if values too large
         break;
      } else {                     // otherwise compute reciprocal
         a[i] = 1.0/a[i];
      }                            // end of if statements
   }                               // end of for loop
   return;                         // return to calling routine (optional)
}                                  // end of function
```

Example: Dot Product (Matlab)

```
% Description: Computes the
% dot product of two vectors

% clear memory
clear

% declare variables to be used
n = 10000000;
a = zeros(n,1);
b = zeros(n,1);
i = 0;
sum = 0;
stime = 0;
ftime = 0;

% initialize a and b
for i=1:n
  a(i) = 0.001*i/n;
  b(i) = 0.001*(n-i)/n;
end
```

```
% start timer
stime = cputime;

% compute dot-product
sum = 0.d0;
for i=1:n
   sum = sum + a(i)*b(i);
end

% stop timer
ftime = cputime;
rtime = ftime-stime;

% output result and runtime
fprintf('dot-prod = %g',sum);
fprintf(' runtime = %g',rtime);

% end program
```

Example: Dot Product (C++)

```
// Description: Computes the
// dot product of two vectors

// Inclusions
#include <stdlib.h>   // new, delete
#include <time.h>     // clock()
#include <stdio.h>    // printf()

// main routine
int main(int argc, char* argv[]) {

  // declarations
  int i, n;
  double *a, *b, sum=0.0, rtime;
  clock_t stime, ftime;

  // ensure an argument was passed
  if (argc < 2) {
    printf("Error: requires 1 arg\n");
    return 1;
  }

  // set n as input arg, ensure positive
  n = atoi(argv[1]);
  if (n < 1) {
    printf("Error: %i < 1\n", n);
    return 1;
  }

  // allocate the vectors
  a = new double[n];
  b = new double[n];

  // initialize vector values
  for (i=0; i<n; i++) {
     a[i] = (0.001 * (i+1.0)) / n;
     b[i] = (0.001 * (n-i-1.0)) / n;
  }

  // compute dot-product
  stime = clock();
  for (i=0; i<n; i++)
     sum += a[i]*b[i];
  ftime = clock();
  rtime = ((double) (ftime - stime))
          / CLOCKS_PER_SEC;

  // output result and runtime
  printf("  length = %i\n",n);
  printf("dot-prod = %.16e\n",sum);
  printf("run time = %.2e\n",rtime);

  // delete vectors and return
  delete[] a;
  delete[] b;
  return 0;
} // end main
```

Example: Computing π (Matlab)

```
% Description: Computes pi via
% numerical integration
%   pi = 4*int_0^1 1/(1+x^2) dx
% using the midpoint rule over n
% equal subintervals

% clear memory
clear

% declare variables to be used
n=0; i=0; h=0; x=0; f=0; a=0;
stime=0; ftime=0; rtime=0;
my_pi=0; err=0;

%  set integrand function handle
f = @(a) 4 / (1 + a*a);

% input number of intervals
n = input(...
  '# of intervals (0 quits):');
if (n < 1)
   return
end
```

```
%  start timer
stime = cputime;

% set subinterval width
h = 1/n;

% perform integration
my_pi = 0;
for i=1:n
   x = h*(i - 0.5);
   my_pi = my_pi + h*f(x);
end
err = pi - my_pi;

% stop timer
ftime = cputime;
rtime = ftime-stime;

% output result and error
fprintf('  my pi = %.16e',my_pi);
fprintf('true pi = %.16e',pi);
fprintf('  error = %g',err);
fprintf('runtime = %g',rtime);
```

Example: Computing π (C++)

```
/* Description: computes pi via
   numerical integration
    pi = 4*int_0^1 1/(1+x^2) dx
 using the midpoint rule over n equal
 subintervals */

// Inclusions
#include <stdlib.h>   // new, delete
#include <time.h>     // clock()
#include <iostream>   // cin, cout
using namespace std;  // no "std::"

// Prototypes
inline double f(double a) {
   return (4.0 / (1.0 + a*a)); }

// main routine
int main(int argc, char* argv[]) {

  // declarations
  int i, n;
  double h, x, my_pi=0.0, rtime; err;
  double pi=3.14159265358979323846;
  clock_t stime, ftime;

  // input the number of intervals
  cout <<
     "# of intervals (0 quits):\n";
  cin >> n;
  if (n<1)  return 1;

  // start timer
  stime = clock();

  // set subinterval width
  h = 1.0 / n;

  // perform integration
  for (i=0; i<n; i++) {
    x = h * (i + 0.5);
    my_pi += h * f(x);
  }
  err = pi - my_pi;

  // stop timer
  ftime = clock();
  rtime = ((double) (ftime - stime))
          / CLOCKS_PER_SEC;

  // output computed value and error
  cout << "  my pi = " << pi << std::endl;
  cout << "true pi = " << pi_true << endl;
  cout << "  error = " << err << endl;
  cout << "runtime = " << rtime << endl;

  return 0;
} // end main
```

Example: Equilibrium Chemistry

This example computes equilibrium chemical densities at multiple spatial locations, given a random background temperature field, using a simple damped fixed-point iteration as the nonlinear solver.

Key Topics:

- Calling external functions
- Call by reference vs call by value
- Standard math library
- Control structures

Routines available in Matlab, C++, C, Fortran90.

Example: 2D Advection

This example sets up and evolves the 2D first order wave equations in time, using a regular staggered spatial discretization and an explicit "leapfrog" time discretization.

Key Topics:

- File input/output
- Header files, macros
- "Flattened" 2D → 1D array data structures
- Nontrivial loop structures
- "() ? () : ()" conditional operators
- Matlab/Python visualization of simulation output data

Routines available in Matlab, C++, C, Fortran90.

Example: Arithmetic Vector Class

This example creates a simple arithmetic vector class in C++, using a contiguous 1D data layout, and defines “Matlab-like” vector operations on them. It then sets up a simple main() routine to create and test these vectors/operations.

Key Topics:

- Class: definitions, operations, public vs. private data
- Object-oriented computing
- Unit-testing

Available only in C++, though also possible in Fortran90.

Typical Errors

Some typical errors in C++ codes include:

- Accidental declaration of constants with the wrong type:

```
vol = 1/3*pi*r*r*r;   // may first compute 1/3=0
```

- scanf() and fscanf() with wrong type:

```
long int n;              // n is long int, so %li is required
fscanf(fid,"%i",&n);     // but %i will read incorrectly
```

- The compiler makes no effort to ensure that you reference array entries correctly:

```
double* x = new double[10];    // will compile, but at runtime will
x[11] = 2.0;                   // (a) seg-fault or (b) corrupt memory
```

- Be careful about punctuation, since all characters in a program are relevant. A misplaced , or . or ; or : could mean the difference between a functioning program, compiler errors, or more insidious run-time problems.
  - `printf('hello');` vs `printf("hello");`
  - Forgetting a semicolon at the end of a line
  - Starting an array at 1 instead of 0 (and/or ending one entry too late)