CS 294-73
Software Engineering for Scientific Computing

Lecture 5:
More C++, more tools.

Let’s go back to our build of `mdArrayMain.cpp`

```
clang++ -DDIM=2 -std=c++11 -g mdArrayMain.cpp DBox.cpp -o mdArrayTest.exe
```

This is doing three things.

- Running the C preprocessor on the .cpp files.
- Translating the .cpp files into object code (compiler).
- Linking the functions defined in the .o files that are needed, starting with `main`, into an executable file (linker).

Each of this functions could have been invoked separately, by changing the inputs given to the compiler on the command line.

- You can run the C preprocessor by itself and save the generated output (You won’t have a reason to do that in this class).
- You can run the compiler to generate object files, using the –c flag (“separate compilation”).

  ```
  clang++ -c -g -I. -DDIM=2 Box.cpp -o Box.o
  ```

- You can link a collection of files that have already been compiled.

  ```
  clang++ -std=c++11 -o mdArrayTest2D.exe ./DBox.o  ./PowerItoI.o  ./
  mdArrayMain.o
  clang++ -std=c++11 -g -o mdArrayTest2D.exe ./DBox.o  ./PowerItoI.o  ./
  mdArrayMain.cpp
  ```

The C proprocessor -

`# ...` – indicates a preprocessor command. The preprocessor has limited text manipulation capabilities.

```
#include <iostream>
#include "Point.H"
#include "DBox.H"
#include "RectMDArray.H"
#include "WriteRectMDArray.H"
using namespace std;
int main(int argc, char* argv[])
{...
}
```

`#include...`: take the contents of the file and copy here.

`“...”` – look for these files in the directories given by `–I` in the command line.

`<...>` - the system knows where to find these (C++ std files).

Point.H

```
#ifndef POINT_H // what is this ? (ans: include guard)
#define POINT_H
#include <iostream>
#include <array>
using namespace std;
...
class Point
{
public:
  inline Point();
  inline Point(const int a_tuple[DIM]);
  inline Point(const array<int,DIM> a_tuple);
  inline Point(const Point& a_pt);
...
}
#endif
```

DBox.H

```
#ifndef _DBOX_H_
#define _DBOX_H_
#include "Point.H”
#include <iostream>
class DBox
{
Public:
}
#endif
```

Both `Box.H` and `Point.H` are included in `mdArrayMain.cpp.` The include guards keep `Point.H` from being included twice (`_POINT_H_` is already defined the first time `Point.H` is included, so `ifndef _POINT_H_` is false by the time it gets to the include in `DBox.H` .

What happens if you don’t do that?

Other C preprocessor commands

```
#define DIM 3
#define OPTIMIZE
#ifdef OPTIMIZE
#endif
```

Also:

```
#ifdef FOO
#if DIM==3
#if DEBUG
#if 0
```

0 is false, 1 is true. Easy way to temporarily delete / undelete sections of code.

- Definitions can be overwritten in the compile line.
  ```
  clang++ -D DIM=2 –D DEBUG=FALSE –U OPTIMIZE –D FOO
  ```
- Can also use the preprocessor to define functions. Again, something you will not be doing in this class (the preprocessor knows nothing about types, legal expressions – it is just manipulating text).
- Newline ends a preprocessor command, but can sting multiple lines together with \

Doxygen (www.doxygen.org)

- Takes comments in header files and turns them into html and latex documents.
- `/// Short description.`
- `/** ... */ Extended description.`
- Knows C++ .
- Warning: defaults many not include our file naming conventions e.g. `*.H` for headers.

Doxygen Demo

C++ is a *Strongly Typed* language

- Strong refers to the fact that
  - a) there is checking being done
  - b) it happens at compile time
- Typed means that all mixtures of data types has a well defined programmatic interpretation. Usually, most type mixing is just disallowed.
- The following program gets compiler errors, what errors?

```
int main(int argc, char* argv[])
{
  float h;
  Vector v;
  float error = v.norm(1);
  return result;
}
```

compilation 1

```
demoBuild1.cpp: In function ‘int main(int, char**)’:

demoBuild1.cpp:4: error: ‘Vector’ was not declared in
  this scope

demoBuild1.cpp:4: error: expected `;' before ‘v’

demoBuild1.cpp:5: error: ‘v’ was not declared in this
  scope

demoBuild1.cpp:6: error: ‘result’ was not declared in
  this scope

make: *** [demoBuild1.o] Error 1
```

- The compiler doesn’t know what these strange strings mean. It is telling you that there needs to be some declaration of what `Vector` means.

- It’s telling you that you need to declare your variables

Compiler Errors

- A *compiler error* indicates something that **must** be fixed before the code can be compiled.
- **Examples**:
  - You forget a semi-colon (;) at the end of a statement and the compiler reports: `somefile.cpp:24: parse error before` `` ` ``*`something`*`'`
  - You miss a closing } in your code: `unexpected end of file`
- Always remember to fix the first few errors or warnings, since they may be causing all the rest. Compiler messages usually list the file and line number where a problem occurs. Nonetheless, errors often occur on the lines prior to what the error message lists. Especially check the line immediately preceding where the error message indicates.
- Finally, note that some compilers may choose to call something an *error* while others may just call it a *warning* or not complain at all.

Compiler Warnings

- *A compiler warning* indicates you've done something bad, but not something that will prevent the code from being compiled.
- You should fix whatever causes warnings since they often lead to other problems that will not be so easy to find.
- **Example**: Your code calls the pow() (raise to a power) library function, but you forgot to include <cmath>.
  - Because you've supplied no prototype for the pow() function (its in <cmath>), the compiler warns you that it assumes pow() returns an int and that it assumes nothing about pow()'s parameters:
- somefile.cpp:6: **warning**: implicit declaration of function `int pow(...)' This is a problem since pow() actually returns a double. In addition, the compiler can't type-check (and possibly convert) values passed to pow() if it doesn't know how many and what type those parameters are supposed to be.
- **Note:** The compiler will label warnings with the word *warning* so that you can distinguish them from errors.
- You would be amazed at how clever a compiler can be about trying to consider an error to be just a warning

Compiling and Emacs

You should do all your editing / compiling / debugging inside of Emacs.

- It understands C++ and the build process, and provides tools for you do perform the compile / bug fix cycle very fast.
- It is available for all platforms, and looks the same on all platforms.
  - Different flavors of emacs: aquamacs (for macs), xemacs.

What’s contained in a .o file?

```
>nm DBox.o
>nm DBox.o | c++filt
```

Linker Errors

- If you receive a linker error, it means that your code compiles fine, but that some function or library that is needed cannot be found. This occurs in what we call the *linking stage* and will prevent an executable from being generated. Many compilers do both the compiling and this linking stage.

- **Example 1**: You misspell the name of a function (or method) when you declare, define or call it:

```
void Foo();
int main() {
        Foo();
        return 0;
}
```

void foo() { // do something } so that the linker complains:

- *somefile*.o(*address*): undefined reference to \`Foo(void)' that it can't find it.

Streams

```
#include <iostream>
#include <fstream>
```

iostream provides you access to screen I/O. You just use it.

```
ofstream oFile;
oFile.open(a_string);
for (int k = 0; k < a_dbx0.sizeOf();k++)
 {
  Point pt = a_dbx0.getPoint(k);
  double x = pt[0]*dx + .5*dx;
  oFile << x << " " ;
  for (int icomp = 0; icomp < NUMCOMPS;icomp++)
   {
    oFile << a_U(pt,icomp) << " ";
   }
  oFile << endl;
 }
oFile.close();
```

Streams

```
#include <fstream>
```

fstream is for file I/O, same rules as `cout, cin`

```
...
  ofstream oFile; // a stream type, analogous to cout.
  oFile.open(a_string);
  for (int k = 0; k < a_dbx0.sizeOf();k++)
    {
      Point pt = a_dbx0.getPoint(k);
      double x = pt[0]*dx + .5*dx;
      oFile << x << " " ;
      for (int icomp = 0; icomp < NUMCOMPS;icomp++)
        {
          oFile << a_U(pt,icomp) << " ";
        }
      oFile << endl;
    }
  oFile.close();
...
```

In the file named `a_string`

```
...
0.164062 1 1 0.75
0.179688 1 1 0.75
0.195312 1 1 0.75
0.210938 1 1 0.75
0.226562 1 1 0.75
0.242188 1 1 0.75
0.257812 1 1 0.75
0.273438 1 1 0.75
0.289062 1.00001 1.00001 0.750003
0.304688 1.00005 1.00005 0.750026
0.320312 1.00028 1.00028 0.750138
0.335938 1.00098 1.00098 0.750489
0.351562 1.00266 1.00266 0.751331
0.367188 1.00598 1.00598 0.752992
0.382812 1.01161 1.01161 0.755807
0.398438 1.02004 1.02004 0.760021
0.414062 1.03135 1.03135 0.765673
...
```

Let’s look at more documentation.

VisIt Demo (visit.llnl.gov)

Laplacian on a Rectangle

$$\phi : [0,1] \times [0,1] \rightarrow \mathbb{R}$$

$$\Delta \phi \equiv \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2}$$

Discretize using finite differences.

$$\phi^h : \Omega^h \rightarrow \mathbb{R} \ , \ \Omega^h = [0, \dots, N] \times [0, \dots, N]$$

$$h = \frac{1}{N} \ , \ \phi^h_{i,j} \approx \phi(ih, jh)$$

.

$$(\Delta^h \phi^h)_{i,j} \equiv \frac{1}{h^2}(\phi^h_{i+1,j} + \phi^h_{i-1,j} + \phi^h_{i,j+1} + \phi^h_{i,j-1} - 4\phi^h_{i,j})$$

$$\Delta^h \phi^h : \Omega^h_0 \rightarrow \mathbb{R} \ , \ \Omega^h_0 = [1, \dots, N-1] \times [1, \dots, N-1]$$

Poisson's Equation

Want to solve

$$\Delta\phi = \rho$$

$$\phi, \rho : [0,1] \times [0,1] \rightarrow \mathbb{R}$$

$$\phi(x,0) = \phi(x,1) = \phi(0,y) = \phi(1,y) = 0$$

(we will be solving Poisson's equation in many different guises throughout the course)

Discretized form

$$\rho^h_{i,j} = \rho(ih, jh)$$

.

$$\Delta^h \phi^h = \rho^h \text{ on } \Omega^h_0$$

$$\phi^h_{0,j} = \phi^h_{N,j} = \phi^h_{i,0} = \phi^h_{i,N} = 0$$

Poisson's Equation

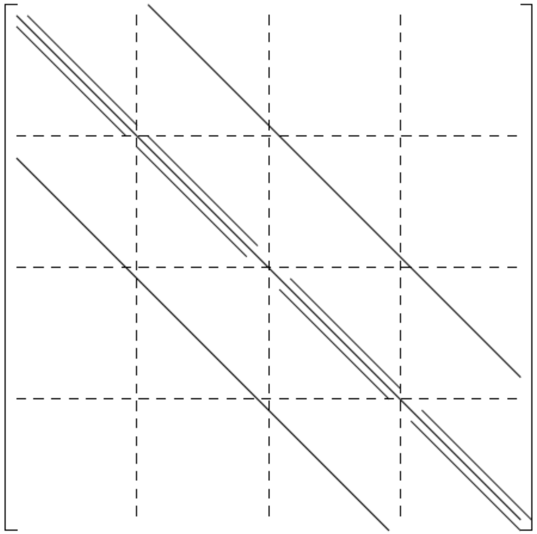
Can be written as a matrix equation

$$Au = f$$

$$u_{i+(N+1)j} = \phi_{i,j}$$

$$\begin{aligned} f_{i+(N+1)j} =& \rho_{i,j} \text{ on } \Omega_0^h \\ =& 0 \text{ on the boundary} \end{aligned}$$

$A =$



(N-1)^2 x (N-1)^2 matrix, nonzeroes along the inner tridiagonal, and two outer sub / super diagonals.

- Banded solve: O(N^3) operations.
- Nested Dissection: O(N^2 log N) operations, but special to this problem.

Point Jacobi Iteration

Motivation: to solve $Au = f$ we compute it as a steady-state solution to an ODE.

$$\frac{d\tilde{u}}{dt} = A\tilde{u} - f$$

If all of the eigenvalues of *A* are negative, then

$$lim_{t\rightarrow\infty}\tilde{u}(t) = u$$

Point Jacobi: use forward Euler to solve ODE.

$$\tilde{u}^{l+1} = \tilde{u}^{l} + \mu\ \left(A\tilde{u}^{l} - f^{l}\right)$$

Stop when the *residual* has been reduced by a suitable amount.

$$||A\tilde{u}^{l} - f|| \leq \epsilon||f||$$

Point Jacobi Iteration

Advantages:

- Simplicity: you don’t have to form, store the matrix, just apply the operator.
- Generality: only depends on the eigenvalues having negative real part in order to converge (or positive real part – just can’t have change of sign). Typical of operators arising from discretizing elliptic, parabolic problems.

Disadvantage: converges slowly.

Residual-Error Analysis

- Look at the equation for the error

$$\delta^l = \tilde{u}^l - u$$

$$A\delta^l = R^l = A\tilde{u}^l - Au = A\tilde{u}^l - f$$

$$\delta^{l+1} = \delta^l + \mu R^l = \delta^l + \mu A \delta^l$$

- If μ is less than the -1/(minimum eigenvalue of *A)*, then $||\delta^{l+1}|| < ||\delta^l||$. To see this, expand δ^l in the eigenvectors of *A,* and look at the evolution of each eigenmode separately. If λ is the corresponding eigenvalue, then $0 < 1 + \mu\lambda < 1$ and the amplitude of the corresponding eignenmode is decreased by that factor at each iteration. For Laplacian, eigenvalues range from -1 to -O(1/h^2), so that convergence of point relaxation is slow:
  - $\mu = O(h^2)$
  - for small eigenvalues, $1 + \mu\lambda = 1 - Ch^2$
  - But this is still a good starting point for better iterative methods.