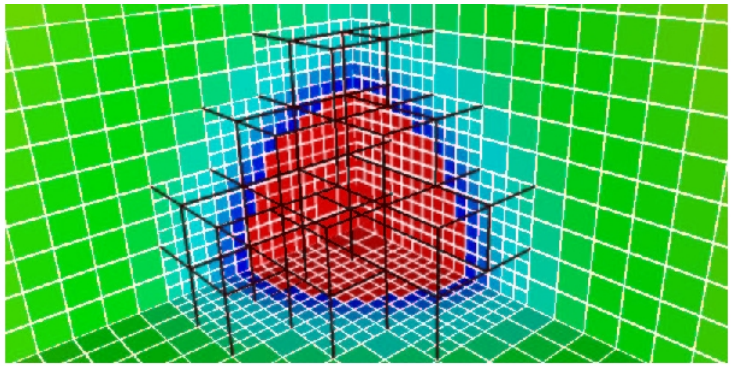
CS 294-73
Software Engineering for Scientific Computing

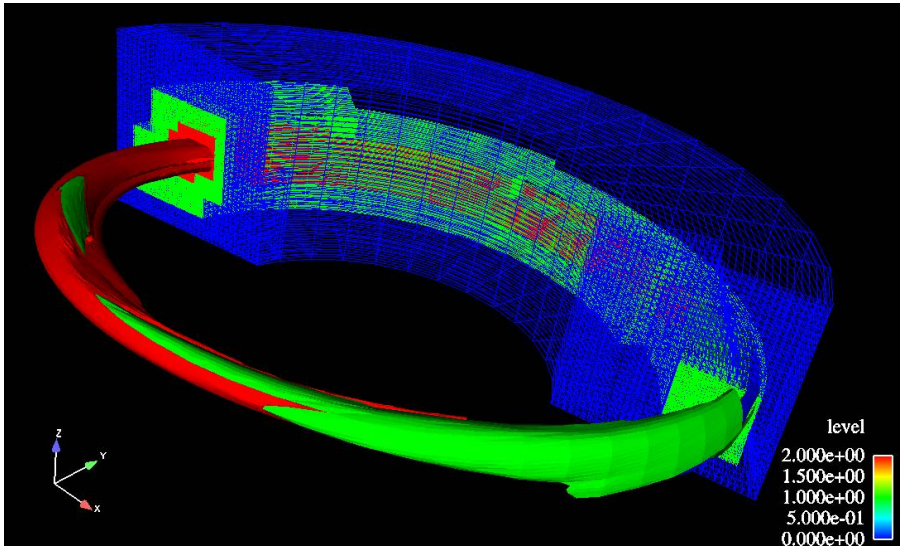
Lecture 4:
Structured Grids;GMake

Structured Grid Calculations



- Numerical solution represented on a hierarchy of nested rectangular arrays.
- Three kinds of computational operations:
  - Local stencil operations on rectangles. Explicit methods, iterative solvers.
  - Irregular computation on boundaries.
  - Copying between rectangles.

Algorithmic Characteristics



- O(1) flops per memory access (10 - 100's).
- Codimension one irregularity.
- Multiphysics complexity: many different operators acting in sequence on the same collection of state variables (the operators may contain state for performance reasons.)
- Irregular computation combined with irregular communication.

Simplest Case: 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.

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

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

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

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

- Regular data access – lots of stride 1 access if you set it up correctly. Computing locations of data simple.
- Mathematical properties of these discretizations well-understood.

Arrays Over a Single Rectangular Grid.

C++ does not have multidimensional arrays with run-time specification of dimensions. Want to build a C++ class to provide us with that feature specifically for use in rectangular-grid discretizations of PDE.

- Make memory management automatic through suitable definition of constructors / destructors.
- Modularity / separation of concerns.
- Anticipate generalization to unions of rectangles.
- Enable implementations that are independent of dimension, in the sense that it appears as a compile-time parameter.

Representing Data on a Rectangle

```
template <class T,unsigned int NCOMP=1> class
  RectMDArray

{

public:

/// bunch of member functions here


private:

DBox m_box; // a separate class to represent range

            // of indices over which array is defined.

T* m_data; // contiguous block of data.

}
```

Representing Data on a Rectangle

```
Class Point

{

public:

/// bunch of member functions here

private:

std::array<int,DIM> m_foo; // integer tuple.

/* DIM is a compile-time constant, given by the
  dimensionality of the space. */
```

Representing Data on a Rectangle

```
Class DBox

{

public:

/// bunch of member functions here


private:

Point m_lowCorner; // low corner (e.g. (0,0)

Point m_highCorner; // low corner (e.g. (N,N)
```

Representing Data on a Rectangle

Why make a separate Classes (`Point, Box`) to represent the range of indices over which the array is defined ?

- Facilitates writing dimension-independent code.
- Operations occur on subsets of the grid: applying the operator, boundary conditions. These subsets can be computed using member functions of `Box` (set calculus).
- Building block for defining data on unions of rectangles.

A walk through `mdarrayMain.cpp`

```
#include <stream>
int main(int argc, char* argv[])
{
Point lowCorner,highCorner;
cout << "input size in each direction (" << DIM <<" ints)" << \endl;
for (int i = 0; i < DIM; i++)
    {
      int length;
      cin >> length;
      highCorner[i] = size-1;
      lowCorner[i] = 0;
    }
  double h = 1./(hi[0]);
  DBox D(lowCorner,highCorner);
  DBox D0 = D.grow(-1);
  RectMDArray<double> phi(D);
  RectMDArray<double> LOfPhi(D0);
```

Grid size coming in from terminal input

Define mesh spacing.

Define Ω^h

Define Ω_0^h by shrinking Ω^h

Define ϕ^h as an array on Ω^h

Define data holder for $\Delta^h \phi^h$

A walk through `mdarrayMain.cpp`

```
for (Point p=D.lowCorner();!(p == D.highCorner());D.increment(p))
{
    double val = 1.;
    for (int dir = 0; dir < DIM; dir++)
      {
         val = val*sin(2*M_PI*p[dir]*h);
      }
     phi[p] = val;
}
```

A walk through `mdarrayMain.cpp`

```
for (Point p=D0.lowCorner();!(p==D0.highCorner());D0.increment(p))
 {
   LOfPhi[p] = 0.;
   for (int dir = 0;dir < DIM; dir++)
     {
       LOfPhi[p] += (phi[p+Point::UNIT(dir)] +
              + phi[p-Point::UNIT(dir)]
     }
   LOfPhi[p] -=2*DIM*phi[p];
   LOfPhi[p] /= h*h;
 }
```

C++ Macro `define`

Where did `DIM` come from ?

At the top of `Point.H` , I included the following statement.

```
#define DIM 2
```

Prior to compilation, the C preprocessor replaces every occurrence of `DIM` with 2 (or 3 or 4 or whatever you choose).

We can override this in the compile line.

Clumsy constructions and missing pieces.

```
for (Point p=D.lowCorner();p != D.highCorner();increment(p,D))
 {
    phi[p] = ...; // phi[p] = phi(p,0), 0 = component number.
 }
```

We need to recompute a lot of information, rather than just incrementing an integer. Inside of RectMDArray, indexing is done by computing

```
ind = p[0] + (m_box.highCorner()[0] - m_box.lowCorner()
[0])*p[1];

return m_data[ind];
```

(actually, more complicated if dimension-independent), instead of incrementing by 1 along each row.

This is also a potential performance issue. What order do we traverse the points ? Function call overheads ? Missed opportunities for optimization ?

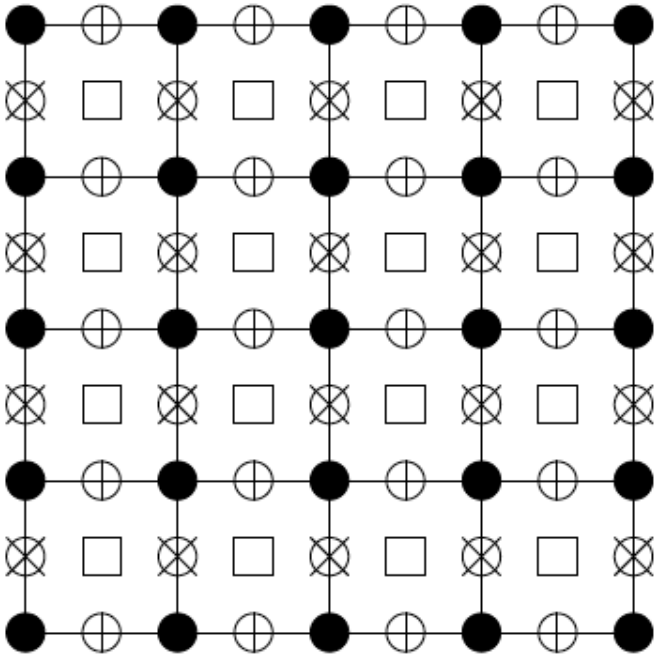
Clumsy constructions and missing pieces.

```
  BoxIterator bi(D0);
  for (bi.begin();bi.notDone();bi++)
    {
       LOFPhi[bi()] = ...;
     }
...
Class BoxIterator
{
  public:
  BoxIterator(const Box& a_box);
  void begin();
  bool notDone() const;
  void operator++();
  Point& operator()();
```

- By storing information in the data members of the iterator can reduce the amount of integer computation used to index into `LOfPhi`.
- What about indexing into arrays with other boxes?
- What can we do for multiple components?

Clumsy constructions and missing pieces.

Different centerings of Boxes:



Vertex (•), cell (□) and face (⊕ and ⊗) sites on a grid

Do you want to handle these by having them as an attribute of Box, or of RectMDArray ?

Other design choices.

Public copy constructors, assignment operators: yes or no ?

- Pros: richer set of operators:

```
RectMDArray<float> A,B,C;

...

A = B + C;
```

- Cons: for large objects, can lead to memory bloat that is not obvious or easily controlled by the user.

Other design choices.

Strong construction vs. default + define.

```
// Strong construction
Box B;
Box C(lo,hi);
B = C;
// Default + define.
RectMDArray<foo> phi(B);
RectMDArray<foo> psi;
psi.define(C);
```

Making aggregates:

```
Box BArray[5];
BArray[0] = B;
BArray[1] = C;
...
```

Default and define:

```
RectMDArray<foo> Phi[5];
Phi[0].define(BArray[0]);
Phi[1].define(Barray[1]);
```

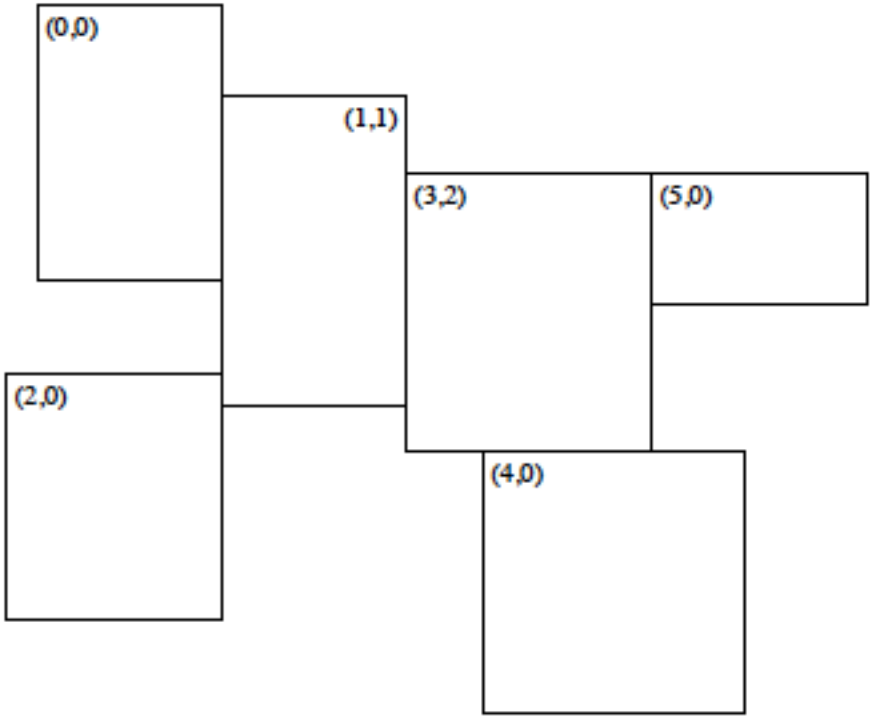
Strong construction: use an array of pointers and `new`:

```
RectMDArray<foo>* Phi[5];
Phi[0] = new RectMDArray(Barray[0]);
Phi[1] = new RectMDArray(Barray[1]);
```

But then you need to make sure you delete.

Beyond a single rectangle.

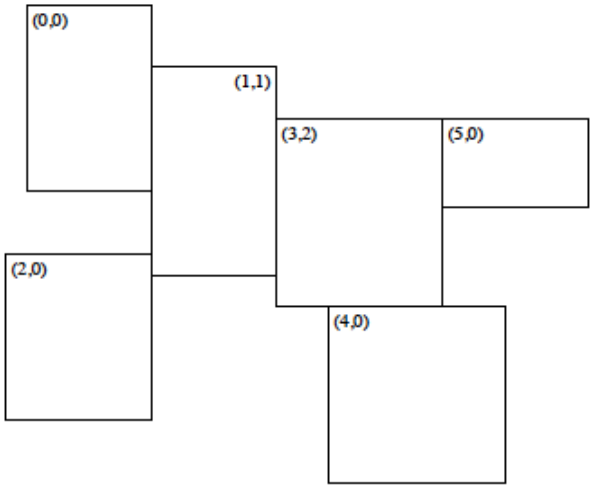
What if you wanted to compute on a domain line made up of several rectangles ?



Can create aggregate data types for representing the region and the data.

Beyond a single rectangle.

```
Class BoxLayout

/* Wrapper for a collection of Boxes
  that form a disjoint covering of
  some region in space. */

Class BLIterator

/* Iterator over the Boxes in a
  BoxLayout. */

Class LevelData

/* Wrapper for a collection of
  RectMDArrays , defined over a
  region given by a DBL. May include
  ghost cell storage. */
```

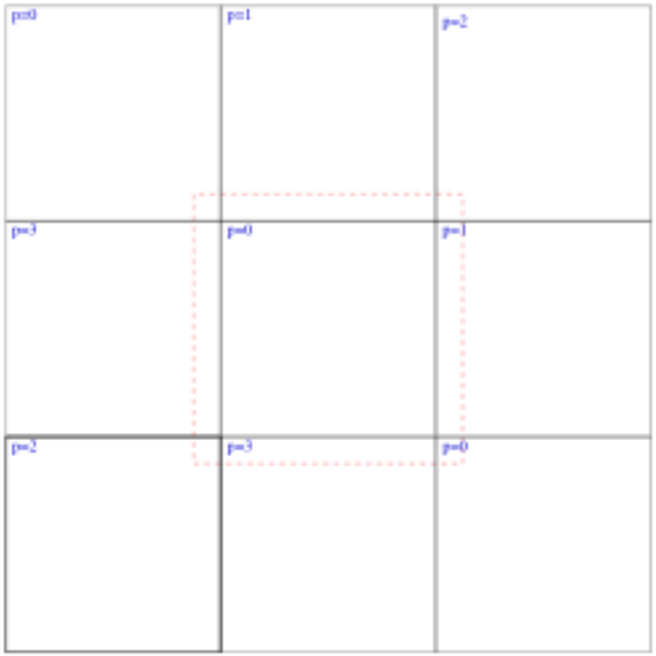


`BoxLayout` is essentially a cell-centered construction – "disjoint covering" = no shared points between the `Boxes` in a DBL. The various other centerings of arrays can be supported, at the expense of redundant data across boxes.

Beyond a single rectangle.

Applying the operator on a union of rectangles.

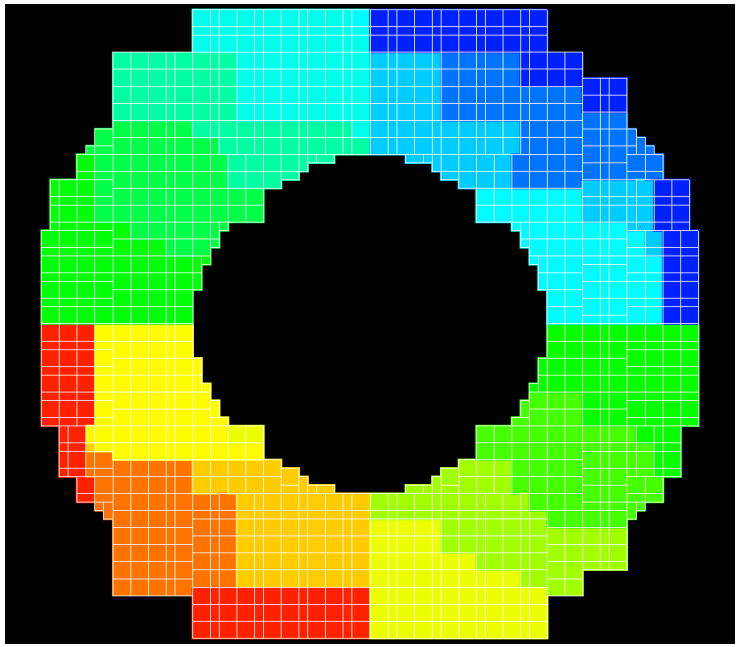
- Get the additional values you need for your stencil from adjacent `RectMDArrays`.
- Iterate over the `RectMDArrays` in a `LevelData`, applying each operator separately.



Beyond a single rectangle.

Design Issues:

- You need to find your neighbors to obtain ghost cell values. How do you do that quickly i.e. no worse than O(log N) per Box (answer: clever sorting, caching results).
- Cost of construction and storage of DBL can be significant, if you store one for every LevelData object. How do you minimize that overhead (answer: sharing).
- What do you do about data that is not cell-centered ? (answer: DBL consist of Boxes that are cell-centered, associated data overlaps.



Development Tools: GNU Make

GNU Make (https://www.gnu.org/software/make/manual)

- A tricky bit of script parsing to manipulate files specialized to work well with compiling code
  - lots of features to let you do simple things simply.
  - complicated things without too much work.
  - almost impossible to figure out what is going wrong.
  - Main purpose: turn a set of source code into a library or executable.
- Only two kinds of objects in a Makefile
  - Variables (lists of strings)
  - Rules
- Only a few kinds of flow control
  - ifeq/ifneq/else/endif
  - No forms or looping available, no jumps, no recursion.
- Most difficulties arising from make are related to
  - Non-trivial variable parsing of the makefile(s)
  - Rules can fire and trigger in non-obvious ways
  - The mysteries of regex

The Two type of Variables in GNU Make

- **Recursively Expanded Variables "="**

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?
all:;echo $(foo)
```

```
> make all
Huh?
```

- Variable is executed at the time it is used in a command
- = means build up a symbol table for this name
- Notice `$`. Like in shell, there is the value 'bar' and the variable named 'bar'

- Good points:
  - Order doesn't matter for = .
  - Can declare a variable as the composite of many other variables that can filled in by other parts of the Makefile
  - `CFLAGS = $(DEBUG_FLAGS) $(OPT_FLAG) $(LIB_FLAGS)`
  - Allows Make to build up sophisticated variables when you don't know all the suitable inputs, or what parts of the Makefile they will come from

```
>make all DIM=3
```

- Bad points:
  - No appending

```
# error, causes infinite loop
CFLAGS = $(CFLAGS) –c
```

  - Future = declarations can clobber what you specified
  - The last = declaration in the linear parsing of a Makefile is the *only* one that matters

- Simply Expanded Variables “:=“, “+=“
  - Immediate mode variable.
  - The variable is assigned it’s value based on the current state of the Makefile parsing
  - No symbol chain is created.
  - Order matters for a series of incremental definitions of a single variable.
  - Specific to GNU Make
- Often just an easier to understand variable.
  - It acts like variables you know in other languages.
  - can use for appending

```
CFLAGS := $(CFLAGS) –c –e –mmx
CFLAGS += –c –e –mmx
```

Rules

targets : *prerequisites* (what does this target need in order to build ?)

[TAB*] recipe*

[TAB] recipe

- prerequisites are also called “sources”

- Simple example

```
clobber.o : clobber.cpp clobber.h config.h
[TAB] g++ -c –o clobber.o clobber.cpp
clob.ex : clobber.o killerApp.o
[TAB] g++ -o clob.ex cobber.o killerApp.o
```

Let's actually use this ...

```
DIM = 2

CXX := clang++

#CXX := g++
```

= comment. Making the compiler a variable makes it easy to switch between Mac and Linux

```
mdArrayTest: GNUmakefile mdArrayMain.cpp DBox.cpp \
  RectMDArray.H RectMDArrayImplem.H DBox.H Point.H \
  PointImplem.H
        $(CXX) -DDIM=$(DIM) -std=c++11 -g mdArrayMain.cpp \
   DBox.cpp -o mdArrayTest.exe
```

The compiler is actually doing three things: running the C preprocessor, compiling the .cpp files, and linking the object files into an executable.

- `-DDIM=$(DIM)` – The preprocessor defines the macro variable `DIM` in all of the source code, overriding local definitions.
- `-std=c++11 -g` – compiler flags.
- `-o mdArrayTest.exe` – executable is the output from the linker (default is `a.out`).

More powerful rules

- Pattern Rules

```
%.o : %.cpp
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
    #Gives a pattern that can turn a .cpp file into a .o file
    # $< = wildcard input (%.cpp), $@ = wildcard output (%.o).
```

- `include, -include` : bringing in other definitions / rules
- `VPATH` : defining where to look for dependencies.

makefile2

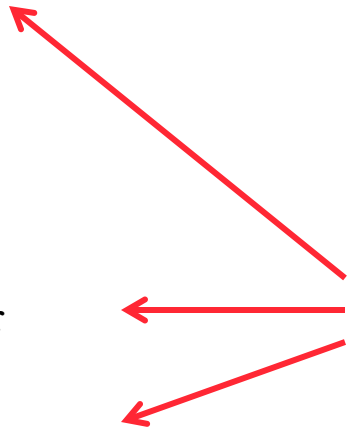
```
# -*- Mode: Makefile -*-
HOME = .
MAKEFILE = makefile2
DIM = 2
CXX := clang++
CFLAGS := -std=c++11 -g
CFLAGS += -I.
CFLAGS += -DDIM=$(DIM)
SRCFILES:= $(wildcard $(HOME)/*.cpp)
OBJS:=$(patsubst %.cpp, %.o, $(SRCFILES))
%.o: %.cpp $(MAKEFILE)
        $(CXX) -c  $(CFLAGS) $< -o $@
        $(CXX) -MM $(CFLAGS) $< > $*.d
mdArrayTest: $(MAKEFILE) $(OBJS)
        $(CXX) $(CFLAGS) -o mdArrayTest.exe $(OBJS)
-include $(OBJS:.o=.d)
```

This tells emacs that the file is a makefile.

I've made this a variable.

makefile3

```
HOME = .
UTILITIES = $(HOME)/utilities
VPATH= $(HOME) $(UTILITIES)/VisitWriter
MAKEFILE = makefile3
DIM = 2
CXX := clang++
CFLAGS := -std=c++11 -O3
CFLAGS += -I. -I$(UTILITIES)/VisitWriter
CFLAGS += -DDIM=$(DIM)
SRCFILES:= $(wildcard $(HOME)/*.cpp $(UTILITIES)/VisitWriter/
*.cpp)
OBJS:=$(patsubst %.cpp, %.o, $(SRCFILES))
%.o: %.cpp $(MAKEFILE)
        $(CXX) -c  $(CFLAGS) $< -o $@
        $(CXX) -MM $(CFLAGS) $< > $*.d
mdArrayTest: $(MAKEFILE) $(OBJS)
        $(CXX) $(CFLAGS) -o mdArrayTest$(DIM)D.exe $(OBJS)
```



Need to tell build process about Visit in three locations

makefile3

```
clean:
        rm -r *.exe $(OBJS:.o=.d) $(OBJS)
-include $(OBJS:.o=.d)
```

What the “make” program does

- Much confusion about make comes from thinking that the Makefile *is* the make program
  - Easy to see. It looks like a shell script.
  - Remember: Makefile is only Variables & Rules
- make:
  - parses *all* of your Makefile
  - builds up variable chains (overriding variables defined on command line)
  - builds up rules database
  - Then looks at what target the user has specified
  - make then attempts to create a chain of rules from the files that exist to the targets specified.
    - recursive “=“ variables in source-target expressions are evaluated
  - Using the date stamp on files discovered in the chain make executes recipes to deliver the target.
    - “=“ variables are evaluated in recipes.

doxygen

- Takes comments in header files and turns them into an html document.
- `/// Short description.`
- `/** ... */ Extended description.`
- Knows C++ .
- Warning: defaults many not include our file naming conventions.