CS 294-73
Software Engineering for Scientific Computing

Lecture 5:
More tools: Gmake, debuggers, git, visualization
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 its 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
    \[
    \text{CFLAGS} := $(\text{CFLAGS}) \text{ –c –e –mmx} \\
    \text{CFLAGS} += \text{ –c –e –mmx}
    \]
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
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 \
Proto_Point.H Proto_PointImplem.H Proto_Box.H \
Proto_BoxImplem.H Proto_BoxData.H Proto_BoxDataImplem.H \$
(CXX) -DDIM=$(DIM) -std=c++11 -g mdArrayMain.cpp \
-o mdArrayTest.exe

• This makefile does the job, but doesn’t scale up to complicated systems.
  • (Demo)
More powerful rules

• Pattern Rules

\%
\.
\.
.o : \%.
.cpp

\$(CXX) -c \$(CFLAGS) \$(CPPFLAGS) $< -o $@

Gives a pattern that can turn a .cpp file into a .o file

$< = wildcard input, $@ = wildcard output.

• include, –include: bringing in other definitions / rules

• VPATH: defining where to look for dependencies.

• Compiler-generated dependencies

  – \$(CXX) –MM \$(CXXFLAGS) $< \$(CPPFLAGS) > $*.d
GNUmakefile (version 2)

CXX=clang++
DIM=2

TARGET:=MDArrayTest.exe
SRC = $(wildcard *.cpp)
OBJ=$(subst .cpp,.o, $(SRC))
DEPENDS:=$(subst .cpp,.d,$(SRC))

CPPFLAGS:=-DDIM=$(DIM)  -I.
CXXFLAGS:= -g -std=c++11

$(TARGET) : $(OBJ)
   $(CXX) -o $(TARGET) $(OBJ)

%.o : %.cpp  (can compile your .cpp files using make).
   $(CXX) -c -o $@ $< $(CXXFLAGS) $(CPPFLAGS)
   $(CXX) -MM $(CXXFLAGS) $< $(CPPFLAGS) > $*.d
clean:
    rm -f *.o *.exe *.d

-include $(DEPENDS)

(Demo)
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

• Invoking 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.
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?

```cpp
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.
Emacs / compiling demo

- Using gdb (g++ compiler) or lldb (clang++ compiler).

- Very similar functionality, but annoyingly different command-line syntax. Cheat sheets are available for both.

- gdb does a better job of interacting with emacs.
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.

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

```c
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.
What’s contained in a .o file?

>nm mdArrayMain.o
>nm --demangle -A mdArrayMain.o

Kinds (examples)
A : Global absolute symbol.
a : Local absolute symbol.
B : Global bss symbol.
b : Local bss symbol.
D : Global data symbol.
d : Local data symbol.
f : Source file name symbol.
T : Global text symbol (function defined in .ccp file)
t : Local text symbol.
U : Undefined symbol (function declared and used in .cpp file, but not defined)

You shouldn’t have any link errors in homework 1 – everything is header files (no *.cpp).
VisIt Demo (visit.llnl.gov)

• Generating VisIt files from your application
  - include Proto_WriteBoxData.H

• Looking at VisIt files.
  - Need to download VisIt from website.

• Generating VisIt files from inside the debugger.
Version Control

Especially for large projects, work does not happen in serial

• Example:
  - Suppose Version 1.0 of your code works fine
  - You begin working on feature A, but realize feature B is more urgent.
  - You want to start working on B with Version 1.0 as a starting point (since you know it works) but don’t want to scrap your work on feature A.

• Other Examples:
  - Two developers need to work on different features separately
  - While editing your code, it gets horribly convoluted; you would like to return to a working version
  - Your project partner changed a bunch of things without documentation
Solutions Offered By Git

- Git is a version control system that solves some of these problems.
  - Save work in snapshots or checkpoints called **commits**.
- Commits are summarized in a log documenting modifications.
- Ability to **branch** workflow and later **merge** branches painlessly.

---

**Schematic Git Workflow**

- Begin Development
- Develop Feature A
- Branch
- Develop Feature B
- Merge
- Continue Development
- Revert Changes to Feature A

represents a commit
Using Git: getting started

• Clone or pull updates from an existing repository:

  • *git clone name@host:repo_name* in this class:

  • *git clone cs294-73@gilman.cs.berkeley.edu:resources* This will create a copy of the repo “resources”

• If you already have a copy of the desired repository, use:*git pull*. This will sync the local directory with the remote repo. Notably, if your local code and the repo version have diverged, git will try to merge them.
Adding and Deleting files

- Add files to the “index” so they will be tracked by git:
  - `git add file_i_just_made.txt` Add a file to the index of the repository.
  - `git delete file_i_dont_want.txt` Delete a file from the index for the repository.
  - `git commit –m “description of changes here”` Commit the changes in the index to the repository. After a commit, all changes are still local. To update the remote repo:
    - `git push origin <branch_name>`
Add / commit / push

• Some notes on add, commit, and push:

• These commands all alter the state of the code in whichever branch you are in (more on this in a bit)

• Generally, it is good practice to group similar changes in a commit (e.g. adding a new piece of functionality or fixing a group of bugs).

• The commit message should be representative and concise (just like commenting your code... which is also good practice)

• In this class, if you try to push to a repo that you shouldn’t, git won’t let you. This is because we are using git with a layer of authentication on top.

• Be discriminating about what you add to the repo:
  - No binaries (*.o, *.exe), or files that you regenerate when building (*.d)
  - No intermediate files from latex (*.aux, *.log).
  - Be very careful about adding while using wildcards (“add *”). You can end up adding git internal files that way, and then you can get in a hopeless snarl.
  - Mac users: MAC OS X doesn’t distinguish between cases. Avoid filenames that are the same except for case (Foo.H, foo.H).
Status of your git repo.

• At any point, you can check the status of your edits since the last commit with:

• **git status**

• You can get a summary of your current edits vis a vis the last commit in the branch using:

• **git diff**

• You may also view the log of previous commits in your current branch using:

• **git log**
Version Control

• An organizational protocol for keeping track of different versions of a project

• Example: You finally get part of your final project for CS294 to work

• Before moving on, you copy all of your code to a separate directory (probably called something like FINALLY_WORKING)

• The backup copy in this example is a version

• We would like to keep track of versions in a more sophisticated way
Branching

• When a git repo is first instantiated, there is one branch: master. For most of your assignments in CS294-73 you will stay on the master branch.

• You can create a new branch with: `git branch branch_name`

• If you aren’t sure which branch you are on, simply type `git branch`

• To switch to a different branch: `git checkout branch_name`
Branching

• Some notes on branching:

• When a new branch is created, its initial state is the last commit in the branch in which it was created

• You can create a branch starting from pretty much any commit in the project tree

• If you have uncommitted changes when attempting to create a new branch, `git` may complain. It’s best to create a new branch right after a commit (i.e. from a clean slate)

• When you switch branches, the files in your local repo will take on the state of that branch.

• There is nothing “special” about the master branch. It’s just the first one in the project.
Merging

- Generally, after a project has branched into parallel versions, you will want to merge them back together. From e.g. branch_A:
  
  - `git merge branch_B` Usually this will be fine, even if changes are made to the same file in both branches
  
  - Occasionally there will be conflicts that `git` can’t resolve. This usually happens when both branches alter the same line of code in different ways.
  
  - To avoid conflicts when working in groups, communicate who is working on what part of the code.
More resources

• Very Basic Tutorial
  http://rogerdudler.github.io/git-guide/

• Interactive Tutorial; not a bad place to start
  https://try.github.io/levels/1/challenges/1

• Fairly comprehensive tutorial. – comes highly recommended.
  https://www.atlassian.com/git/tutorials/

• Then, there is always the google.