For ####'s Sake, Just Use Make

“Those who do not understand their tools are doomed to reinvent them, badly.”
– Old Klingon Proverb

“I love the way you can make up a quote and no-one bothers to check the attribution.”
– Quentin Tarantino

Mike Seymour mike@mikeseymour.co.uk https://github.com/mikeseymour

Share and enjoy!

A common problem

- Problem: My build tool doesn't do what I want
- Solution 1: Use something else
  - Plenty of choice
  - make, gmake, cmake, dmake, automake, nmake, jam, bjam, scons, gradle, waf, doit, rake, aap, ant, ninja, tup, redo, …
  - Which of these will do what you want?
- Solution 2: Learn to use it properly
  - It probably does do what you want

Why make?

- Simple, flexible declarative language to specify dependencies and build rules
- As old as the hills
  - Stable and Just Works (if you know what to do)
  - Plenty of knowledge available
- Portable and widely available
- BUT: quirky syntax, some peculiar concepts
  - Best not to try anything too clever

How does make work?

- Build rules specify
  - A target: what to build
  - Dependencies: what to find or build first
  - A recipe: how to build
- Example

```
hello.o : hello.cpp
    g++ -o hello.o -c hello.cpp
```

- Lots of rules like this are built-in

How does make work?

- Variables

```
CXXFLAGS += -std=c++11 -O3 -W -Wall -Wextra -Werror
```

- Append with `+=`, define with `=` or `:=`
  - Don't worry about the difference for now
- Wildcards and automatic variables

```
%.o : %.cpp ; g++ $(CXXFLAGS) -o $@ -c $<
```

- Functions and substitutions

```
app_srcs := $(shell find src/app -name *.cpp)
apps : $(app_srcs:src/%.cpp=obj/%)
```

Actually, I lied...

- Not “just” make
  - GNU make
  - BASH shell
  - GNU or Clang compiler
  - Or suitable compatible alternatives
  - Available on any sensible build platform
- Make can't do everything by itself
  - Find source files
  - Identify header dependencies

Finding source files

- Tedious to list all the source files ourselves
- Use the shell to find all files beneath a directory

```
all_srcs := $(shell find src -name *.cpp)
```

- Use top-level subdirectories to indicate purpose

  `lib` for all the logic to be linked into a library

  `apps` for programs to be compiled separately and each linked with the library

  `test` for tests to be compiled together, linked with the library, and run as part of the build

  - Maybe subdirectories `unit,integration,...`

Tracking dependencies

- C and C++ sources include headers
  - Might include more headers
  - Need to recompile if any changes
  - Far too error-prone to do this ourselves
- The compiler can tell make about them
  - Flag `-MMD` generates dependency makefiles
  - Flag `-MP` adds rules to deal with deleted headers
- Include these from the makefile if they exist

```
-include $(all_srcs:src/%.cpp=obj/%.d)
```

Minimum viable Makefile (1)

```
CXXFLAGS += -MMD -MP
CXXFLAGS += -Isrc -Isrc/lib -std=c++11 -O3
CXXFLAGS += -W -Wall -Wextra -Werror -g

all_srcs := $(shell find src -name *.cpp)
lib_srcs := $(shell find src/lib -name *.cpp)
app_srcs := $(shell find src/apps -name *.cpp)
tst_srcs := $(shell find src/test -name *.cpp)

-include $(all_srcs:src/%.cpp=obj/%.d)
.PRECIOUS : obj/%.o
```

Minimum viable Makefile (2)

```
all   : test apps
clean : ; @rm -rf obj
apps  : $(app_srcs:src/%.cpp=obj/%)
test  : obj/test/main ; @$<


obj/test/main : $(tst_srcs:src/%.cpp=obj/%.o)
obj/lib.a : $(lib_srcs:src/%.cpp=obj/%.o)
   @ar rcs $@ $^
obj/%.o : src/%.cpp
   @mkdir -p $(dir $@)
   $(COMPILE.cpp) $(OUTPUT_OPTION) $<
obj/% : obj/%.o obj/lib.a
   @$(LINK.cpp) $(OUTPUT_OPTION) $^ obj/lib.a $(LDLIBS)
```

Resources

- Example project using this Makefile

  https://github.com/mikeseymour/JustUseMake
- Catch test framework used by the example

  https://github.com/philsquared/Catch
- RTFM

  https://www.gnu.org/software/make/manual/

  https://gcc.gnu.org/onlinedocs