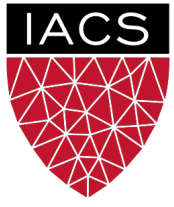
Makefiles

Adapted from Lectures by Chris Simmons

David Sondak



Harvard John A. Paulson School of Engineering and Applied Sciences

IACS Institute for Applied Computational Science

February 21, 2019

What is `make` and Why Use It?

- `make` is a program for building programs
- Used for codes with multiple source files, library dependencies, etc.
- Tries to recompile only things that have changed
- We'll focus on the GNU flavor of `make`
  - Portable
  - Default on Linux
  - `http://savannah.gnu.org/projects/make/`
- BSD is another flavor of `make`

The Basics of `make`

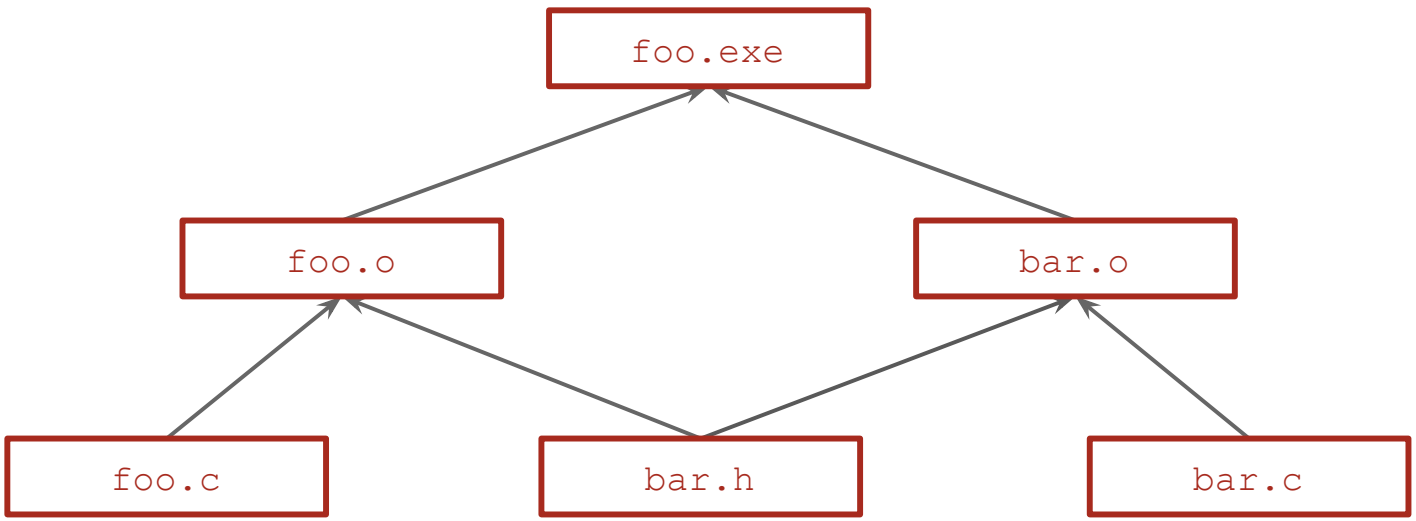
- `make` reads a `Makefile` which includes
  - Files to be created
  - The dependencies of the files
  - Instructions for creation of the files
- `Makefiles` describe a *Directed Acyclic Graph* (DAG) of the dependencies
- `make` uses post-order traversal to work up the dependency graph, building files until the goal file is up-to-date
- `make` only builds files whose dependencies are newer than the goal file

Basic Usage

- Create a file called `Makefile` in the directory that contains your source files
- The convention is uppercase
- Edit `Makefile` with instructions for building your code
- Type `make` to build your program
  - i.e. `$ make`

Basic Example

- Suppose you have three files: `bar.c`, `bar.h`, `foo.c`
- The dependencies are as follows:
  - `bar.c` + `bar.h` → `bar.o`
  - `foo.c` + `bar.h` → `foo.o`
  - `foo.o` + `bar.o` → `foo.exe`



- Using the `gcc` compiler (without `make`), you would do the following:

```
$ gcc -c foo.c
$ gcc -c bar.c
$ gcc -o foo.exe foo.o bar.o
```

Basic Makefile

```
foo: foo.o bar.o
  gcc -o foo.exe foo.o bar.o

foo.o: foo.c bar.h
  gcc -c foo.c

bar.o: bar.c bar.h
  gcc -c bar.c
```

Basic Makefile Rules

```
target: prerequisite
  command
```

- `target` — Files to be created / updated
- `prerequisite` — Files which must be up-to-date before the target can be updated
- `command` — Shell scripts used to update the target

Improving the Makefile

```
CC := gcc
foo: foo.o bar.o
  $(CC) -o $@ $^
foo.o: foo.c bar.h
  $(CC) -c $<
bar.o: bar.c bar.h
  $(CC) -c $<
```

- Assigned a variable with a compiler name
  - Note the := — Immediately evaluates the RHS expression
  - A regular = re-evaluates the expression each time it's used
- $@ — Target of the current rule
- $^ — All the prerequisites of the current rule
- $< — First prerequisite of the current rule

A More Realistic Makefile

```
# Files
EXEC := foo
SRC  := foo.c bar.c
OBJ  := $(patsubst %.c,%.o,$(SRC))
# Options
CC       := gcc
CFLAGS   := -O3
LDFLAGS  := -L/usr/lib
LDLIBS   := -lm
# Rules
$(EXEC): $(OBJ)
    $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^
%.o: %.c
    $(CC) $(CFLAGS) -c $<
foo.o bar.o: bar.h
# phony targets
.PHONY: clean
clean:
    $(RM) $(OBJ) $(EXEC)
```

Discussion of `OBJ:=$(patsubst %.c,%.o,$(SRC))`

- The line `OBJ:=$(patsubst %.c,%.o,$(SRC))` is a pattern subsitution
- Changes every space-separated thing in `SRC` that ends in `.c` to end in `.o`
  - `foo.obj` would not match the pattern
  - Spaces around the commas matter!
- `OBJ` would become `bar.o foo.o`

Discussion of Linking

```
$(EXEC): $(OBJ)
    $(CC) $(LDFLAGS) $(LDLIBS) -o $@ $^
```

- Makes the value of `EXEC` depend on the value of `OBJ`
- `CC` points to the `C` compiler
- `LDFLAGS` are for library paths and `LDLIBS` are for libraries
- `$@` expands to the target of the rule
- `$^` expands to the list of prerequisits

Pattern-based Rules

```
%.o: %.c
    $(CC) $(CFLAGS) -c $<
```

- Creates a rule for creating object files from source files with a similar name
- `CFLAGS` contains any compiler options needed at compiler time (e.g. `-O3`)
- Reminder: `$<` expands the first prerequisite

phony targets

```
.PHONY: clean
clean:
	$(RM) $(OBJ) $(EXEC)
```

- Targets listed as dependendies of `.PHONY` do not reference files
- They are always treated as out-of-date
- Useful for cleaning up a directory

Useful Options

- Typing `make -n -p` can be useful
- `-n` prints commands that `make` would run, but doesn't do anything
- `-p` tells `make` to print out all its rules and variables
- There are other useful options in the manual

Parting Thoughts

- By default, `make` looks for `Makefile` or `makefile`
- To use a different file, type `make -f mymakefilename`
- If you want the manual page, just type `info make`
- You will encounter many `makefiles` in your life of many different characters