Using `make`

```
bash$ make -f <makefileName> target
```

- Defaults:
  - If no `-f` specified, use a file named `Makefile` in current dir
  - If no `target` specified, will use the first one in the file
  - Will interpret commands in your default shell
    - Set `SHELL` variable in makefile to ensure
- Target execution:
  - Check each source in the source list:
    - If the source is a target in the makefile, then process it recursively
    - If some source does not exist, then error
    - If any source is newer than the target (or target does not exist), run `command` (presumably to update the target)

`make` Basics

- A makefile contains a bunch of triples:

```
① target: sources ②
← Tab → command ③
```

  - Colon after target is *required*
  - Command lines must start with a **TAB**, NOT SPACES
  - Multiple commands for same target are executed *in order*
    - Can split commands over multiple lines by ending lines with '\'
- Example:

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


Makefile writing tips

- When creating a Makefile, draw the dependencies!!!!
- C Dependency Rules:
  - `.c` and `.h` files are never targets, only sources.
  - Each .c file will be compiled into a corresponding `.o` file
    - Header files will be implicitly used via `#include`
  - Executables will typically be built from one or more `.o` file
- Good Conventions:
  - Include a 'clean' rule
  - If you have more than one 'final target' have an 'all' rule
  - Put your singular 'final target' or 'all' as the first target.

Writing a Makefile Example

- "`talk`" program (find files on web with lecture slides)

`main.c` `speak.h` `speak.c` `shout.h` `shout.c`

main.c

```
#include "speak.h"
#include "shout.h"

int main(int argc, char** argv) {…
```

speak.c

```
#include "speak.h"
...
```

shout.c

```
#include "speak.h"
#include "shout.h"
...
```