Useful Tool Demo: Makefiles

<gerdela@scss.tcd.ie>

>1 .c file in project

- **`gcc -o myprog file1.c file2.c file3.c`**

- As project gets big - only want to re-compile a file if it has actually changed (speed-up)
  - build system is bad or *optimisation* is struggling. ⟶
  - ~1-5s for <50k LOC. ~Instant for hot-reloading (advanced). ⟶
- De-couple stages of compilation

Build Object Files

- Compile separate object files first
  ```
  gcc -o file1.o file1.c
  gcc -o file2.o file2.c
  gcc -o file2.o file2.c
  ```
- Link together in step 2
  ```
  gcc -o myprog file1.o file2.o file3.o
  ```
- No need to rebuild .o file if its .c didn't change
- Quicker to build

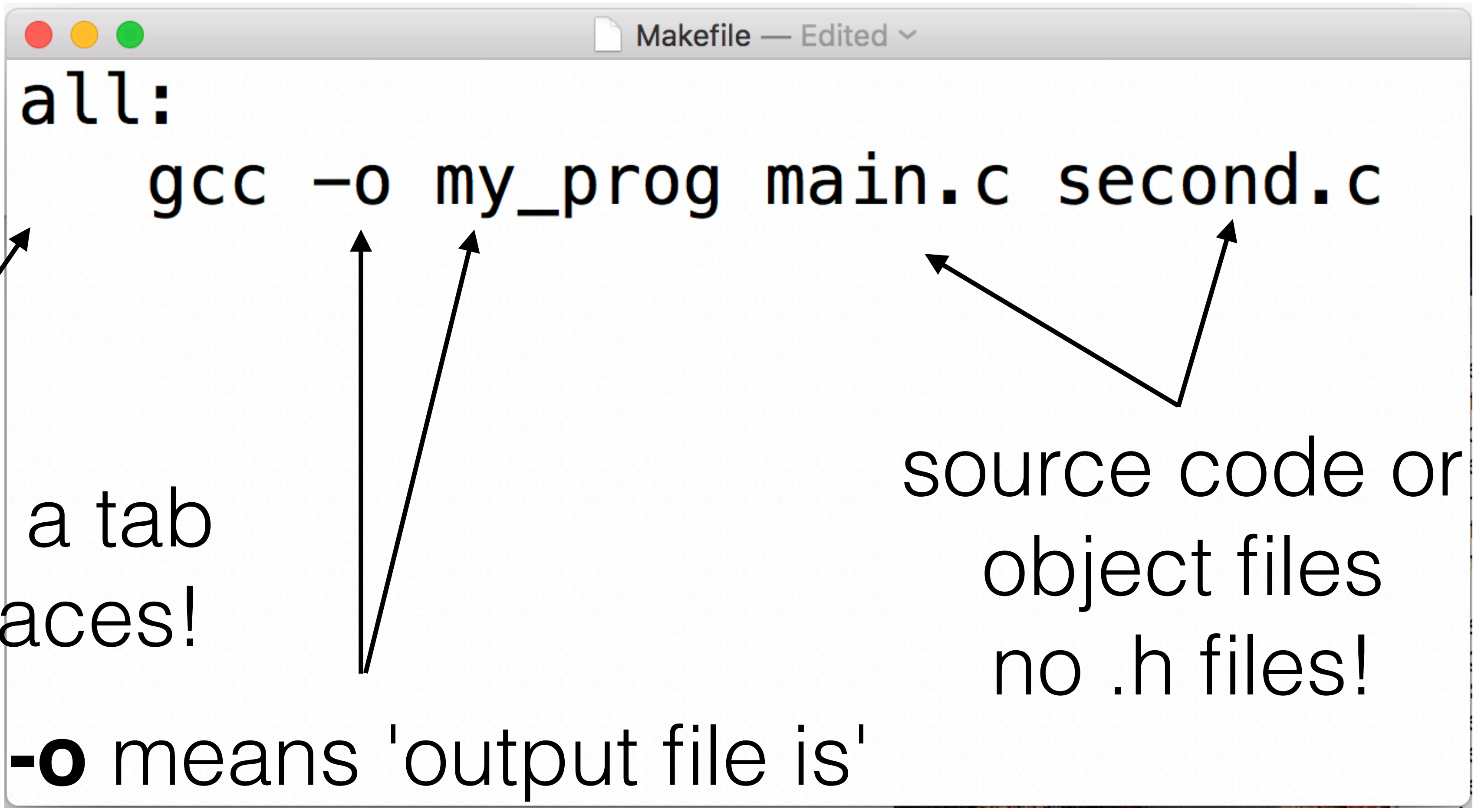
Automate This

- A shell script would be fine
  - e.g. BaSh 'build.sh' or MS *batch* file 'build.bat'
- Some IDEs - maintain own "project"/solution files
- Most open-source and Unix-ish software - has a **Makefile**
- CMake etc are higher-level
  - you maintain a CMake file
  - user runs CMake or CMake-gui to generate whatever Makefile or IDE's project file
  - *CMake a big mess*

Makefile

- Has its own weird little programming language
- Is fairly simple
- Picky about tabs and spaces [*insert groan*]
- Can run any terminal commands
- but mostly designed for our job of building from multiple files
- Execute with 'make' program
  - `$ cd my_projects_folder`
  - `$ make` - this will use the file called 'Makefile'
- Make is usually part of GNU project with GCC and other Unix-like tools.

Simple Makefile



```
all:
	gcc -o my_prog main.c second.c
```

named command

must be a tab - not spaces!

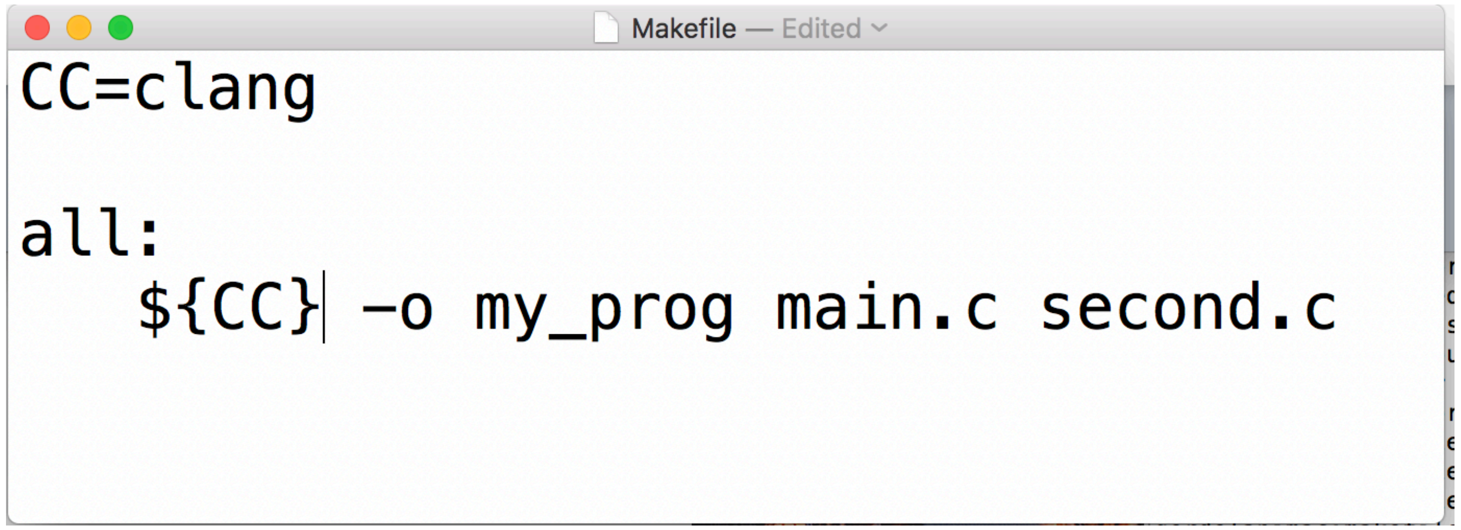
-o means 'output file is' do not leave out the file name you have been warned

source code or object files no .h files!

Then

- Just cd to your folder and type `make`
- looks for a file called *Makefile* (no extension)
- If there is only an '*all*' section it will run that
- To run a specific section:
    - `make all`
    - `make othersectionname`

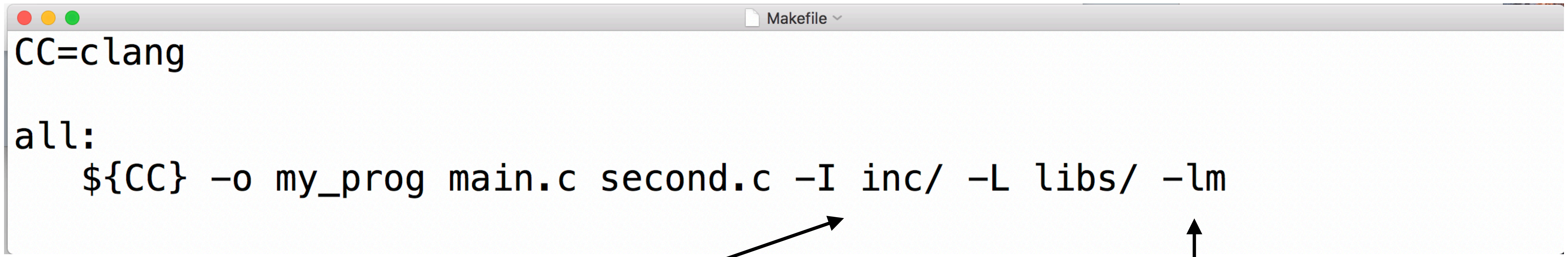
Variables, Flags, Libraries, Include paths



```
CC=clang

all:
    ${CC} -o my_prog main.c second.c
```

define variable

use value of variable

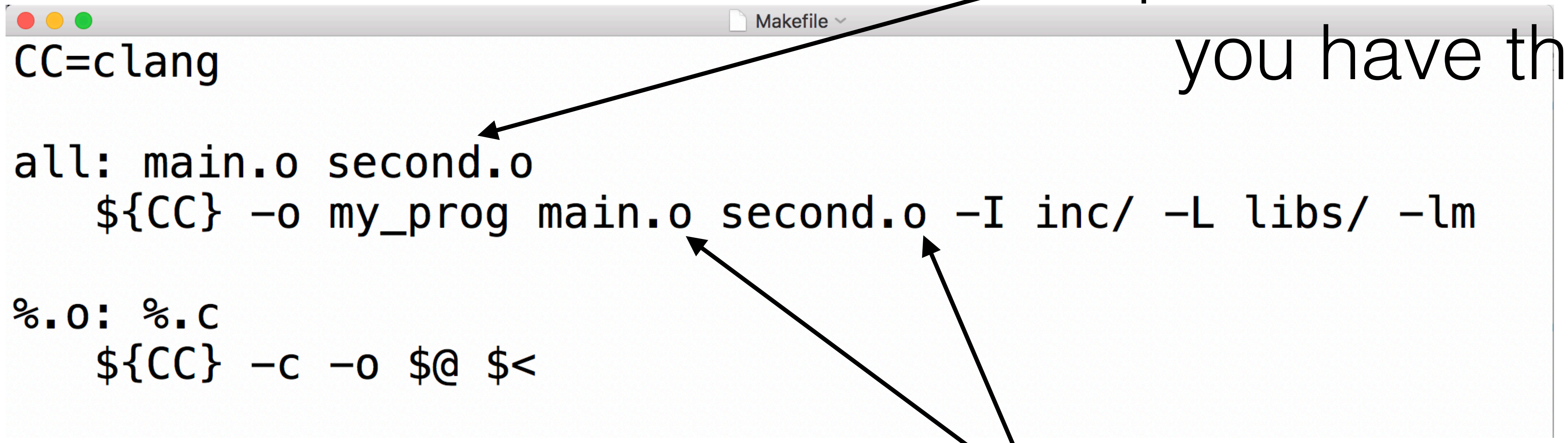


```
CC=clang

all:
    ${CC} -o my_prog main.c second.c -I inc/ -L libs/ -lm
```

"look in these subfolders for headers and libraries"

"look for a file called *libm.dylib*" or *libm.so* or *libm.dll*

Rule for Building Object Files



```
CC=clang

all: main.o second.o
	${CC} -o my_prog main.o second.o -I inc/ -L libs/ -lm

%.o: %.c
	${CC} -c -o $@ $<
```

dependencies - "make sure you have these first"

<u>rule for building a .o</u>

-c means build a .o

$@ is the *thingname*.o

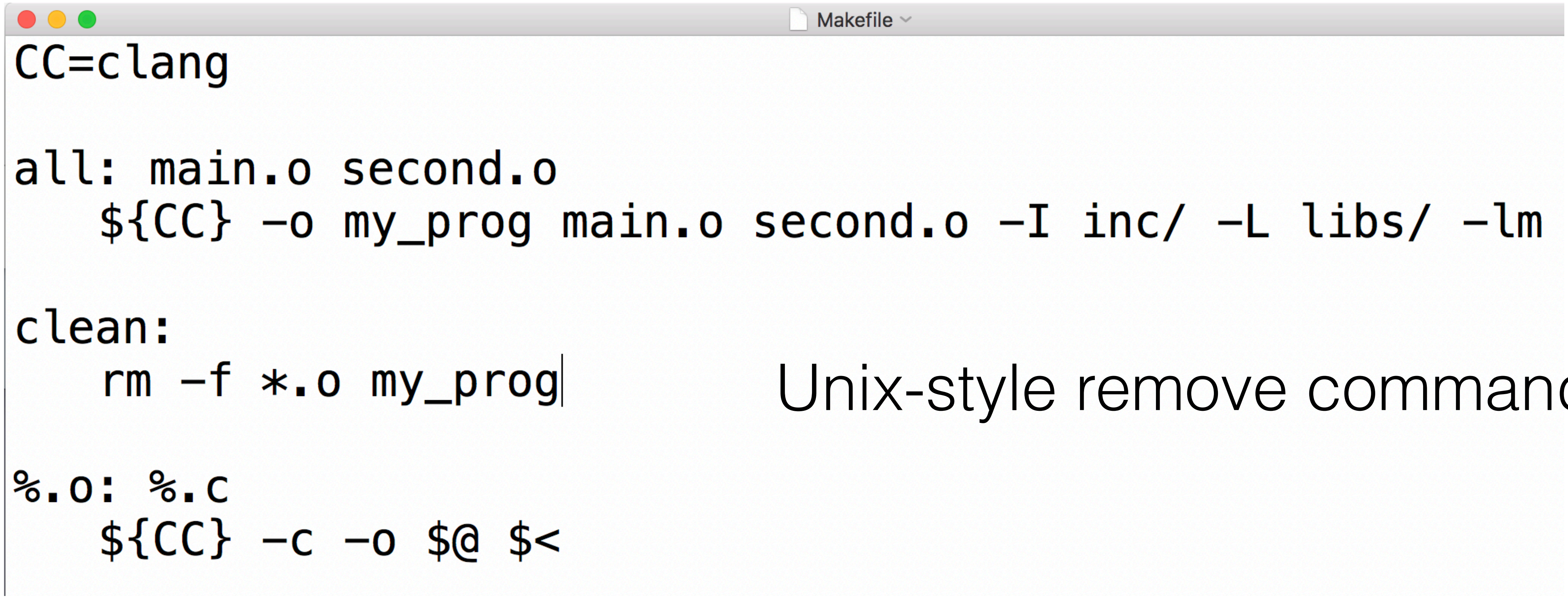
$< is the *thingname*.c

"Only rebuild changed files"

- Try make now. You should get a program and main.o and second.o in your folder.
- What if you delete just one of the .o files and run make again?

Make all, Make clean

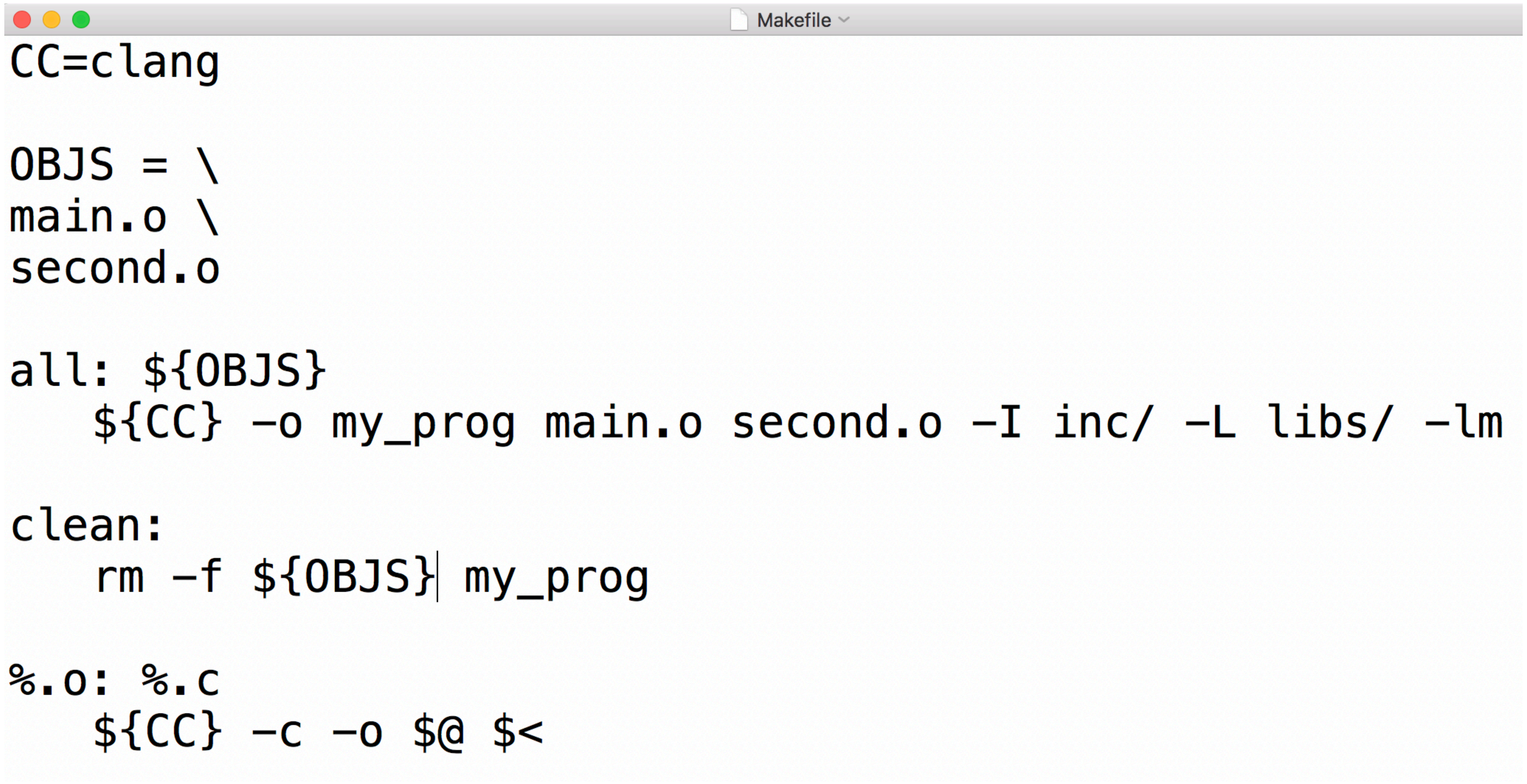
- Sometimes you want a complete rebuild
    - changes in headers, libraries, etc.



```
CC=clang

all: main.o second.o
	${CC} -o my_prog main.o second.o -I inc/ -L libs/ -lm

clean:
	rm -f *.o my_prog

%.o: %.c
	${CC} -c -o $@ $<
```

Unix-style remove command

Source files list



```
CC=clang

OBJS = \
main.o \
second.o

all: ${OBJS}
	${CC} -o my_prog main.o second.o -I inc/ -L libs/ -lm

clean:
	rm -f ${OBJS} my_prog

%.o: %.c
	${CC} -c -o $@ $<
```

\ means 'continue on next line'

GCC/Clang Flags to Know About

- **-o** next string is the output file name
- **-c** compile object file only
- **-g** compile for debug
- **-std=c99** 'compile in C99 mode'. many alternatives.
- -pg compile for profiling and debug
- **-Wall** enable all warnings (do this). maybe also -Wfatal-errors -pedantic
- -O optimise code. others: -Ofast -O3 -O2 -O1
- -DANTON make `ANTON` appear to pre-processor as a `#define`d value
- -fsanitise=… lots of extra checks can be added for leaks/array bounds etc.
- -m64 -arch_x86_64 -mmacosx-version-min=10.11 "only build for these systems"

Advantages

- Much faster builds and linking for larger projects
- Simple(ish) to write, read, and user-modify
    - mostly it's just a list of your .c files
    - e.g. my book code
    - (don't force *your* favourite build system on programmers).
- Everybody knows 'make'

Limitations

- not good at platform switch - I usually end up with (if linking different libraries for each system)
  - *Makefile.win32*
  - *Makefile.linux64* `make -f Makefile.linux64`
  - *Makefile.osx*
- A special language for a build system is insane (but somewhat independent)
- Many IDEs don't use Makefiles

Reference

- GNU make manual
  https://www.gnu.org/software/make/manual/
  reference for flags etc
- `man gcc` or `man clang` or `man make`
- Any make tutorial