Makefile Basics

Ref:
http://nuclear.mutantstargoat.com/articles/make/

https://opensource.com/article/18/8/what-how-makefile

I The very basics

- target
- rule

```
target: <dependencies>
	<rule to build target>
```

The seperator MUST be a tab, NOT whitespace!

Eg.

```
hello: hello.c
	gcc hello.c -o hello -O2 -Wall
```

- If you name your makefile as `Makefile`, then you just need to type `make` to have it get parsed... else, `make -f <makefile-name>`.

- If a target isn't an actual file that will be generated (typically the binary executable), mark it as a dummy or 'phony' like this:

  ```
  .PHONY: clean
  clean:
  	rm -f hello
  ```

II Multiple source files

- you have several src files
- need to compile all into object files and then link the object files into a binary executable
- can specify *libraries* to be (dynamically) linked in via the `-l` option switch;
  - format: `-l<libname>` (just the short name, leave out the 'lib' prefix)

An example

Source files (*.c):
  main.c player.c enemy.c renderer.c vecmath.c image.c mesh.c
Libraries:

libGL, libglut, libpng, libz, libm

```
$ cat Makefile
[...]
mygame: main.o player.o enemy.o renderer.o vecmath.o image.o
mesh.o
     cc -o mygame main.o player.o enemy.o renderer.o \
      vecmath.o image.o mesh.o \
      -lGL -lglut -lpng -lz -lm

main.o: main.c
    cc -c main.c

player.o: player.c
    cc -c player.c

# ... and so on, and so forth ...
```

Difficult to specify each C src file's compilation; easier:

```
myprog: main.o game.o level.o player.o enemy.o renderer.o \
 vecmath.o image.o mesh.o \
     cc -o $@ $^ -lGL -lglut -lpng -lz -lm
```

- **@** is a built-in make variable containing the target of each rule (*myprog* in this case) (specify it as `$@`)
- **^** is another built-in variable containing all the dependencies of each rule (so the list of object files in this case)
- Similarly, although not useful for this particular rule, there is also the **<** variable, which contains only the first element of the dependencies (so `main.o` if we used it in the rule above)

Tips:

- In C, if a function or global variable is to be seen across source files, do NOT mark it as `static`
- If a Makefile variable name is xyz, then $xyz won't work (only the first letter is recognized); specify parentheses around the var name, as $(xyz)
- to print a message within the Makefile, use
  `@echo "my message"`

III Multiple source files, superior syntax – in effect, a makefile for 99% of your programs

"There's one last thing bugging me in the last example. I hate having to type manually the list of object files needed to build the program. Thankfully it turns out we don't need to do that either. See the following example:

```
src = $(wildcard *.c)
obj = $(src:.c=.o)
```

```
LDFLAGS = -lGL -lglut -lpng -lz -lm

myprog: $(obj)
    $(CC) -o $@ $^ $(LDFLAGS)

.PHONY: clean
clean:
    rm -f $(obj) myprog
```

The first line of this example collects all source files in the current directory in the `src` variable. Then, the second line transforms the contents of the `src` variable, changing all file suffixes from `.c` to `.o`, thus constructing the object file list we need.
I also used a new variable called `LDFLAGS` for the list of libraries required during linking (`LDFLAGS` is conventionally used for this usage, while similarly `CFLAGS` and `CXXFLAGS` can be used to pass flags to the C and C++ compilers respectively).

IV Multiple source directories

“Let's say you have some source files under `src/`, other source files under, `src/engine`, and some more source files under `src/audio`. Also, to make it more interesting, let's assume that your code is a mix of C and C++ in any of these directories. Here's a very simple modification which handles this situation:

```
csrc = $(wildcard src/*.c) \
       $(wildcard src/engine/*.c) \
       $(wildcard src/audio/*.c)
ccsrc = $(wildcard src/*.cc) \
        $(wildcard src/engine/*.cc) \
        $(wildcard src/audio/*.cc)
obj = $(csrc:.c=.o) $(ccsrc:.cc=.o)

LDFLAGS = -lGL -lglut -lpng -lz -lm

mygame: $(obj)
    $(CXX) -o $@ $^ $(LDFLAGS)
```

Tip:
Using an ‘if’ conditional

```
LDFLAGS = $(libgl) -lpng -lz -lm

ifeq ($(shell uname -s), Darwin)
    libgl = -framework OpenGL -framework GLUT
else
    libgl = -lGL -lglut
endif
```

V Automatic dependency tracking

"...
There's a way to instruct the compiler (GCC and clang supports this, more compilers might do so too), to generate the necessary makefile fragment (`.d` file) at the same time as it compiles each source file. This is done by passing the `-MMD` option to the compiler.

The absence of the `.d` files during the first make will not be detrimental, because at that point we're doing a full build anyway, and subsequent make invocations will have the necessary makefile fragments included to aid in deciding which source files need to be rebuilt because a header file has changed.

This approach, apart from reducing initial build times, also simplifies our makefile from the previous section slightly, since there's no need for a custom `%.d: %.c` rule for generating the dependency files:

```
src = $(wildcard *.c)
obj = $(src:.c=.o)
dep = $(obj:.o=.d)  # one dependency file for each source

CFLAGS = -MMD     # option to generate a .d file during compilation
LDFLAGS = -lGL -lglut -lpng -lz -lm

mygame: $(obj)
    $(CC) -o $@ $^ $(LDFLAGS)

-include $(dep)   # include all dep files in the makefile

.PHONY: clean
clean:
    rm -f $(obj) mygame

.PHONY: cleandep
cleandep:
    rm -f $(dep)
```