What is a Makefile and how does it work?

If you want to run or update a task when certain files are updated, the `make` utility can comes handy. The `make` utility requires a file `Makefile`(or `makefile`) which defines set of tasks to be executed. You may have used `make` to compile a program from source code. Most of the Open Source projects uses `make` to compile a final executable binary which can then be installed using `make install`.

In this post, we'll explore `make` & `Makefile` using basic and advance examples. Make sure you have `make` installed in your system.

Basic examples

Lets start by printing the classic "Hello World" on the terminal. Create a empty directory `myproject` having a file `Makefile` with below content:

```
say_hello:
    echo "Hello World"
```

Now run the file by typing `make` inside the directory `myproject`. The output will be as below:

```
$ make
echo "Hello World"
Hello World
```

In above example `say_hello` behaves like a function name as in any programming language. This is called the **target**. The **prerequisites** or **dependencies** follow the target. But for the sake of simplicity, we have not defined any prerequisites in the above example. The command `echo "Hello World"` is called the **recipe**. The *recipe* uses *prerequisites* to make a *target*. The target, prerequisites, & recipes together makes a **rule**. To summarize, below is the syntax of a typical rule:

```
target: prerequisites
<TAB> recipe
```

As an example, a target might be a binary file which depends on prerequisites(source files). On the other hand, a prerequisite can also be a target which depends other dependencies. Something like:

```
final_target: sub_target final_target.c
    Recipe_to_create_final_target

sub_target: sub_target.c
    Recipe_to_create_sub_target
```

It is not necessary that the target should be a file. Like in our example, the target can just be a name for the recipe. We call such targets as "Phony targets".

Lets go back to the example above. When `make` was executed, the entire command - `echo "Hello World"` got displayed followed by actual command output. We often don't want that. To suppress echoing the actual command, we have to start `echo` with `@`:

```
say_hello:
    @echo "Hello World"
```

Now try to run `make` again. The output should only be display on the terminal:

```
$ make
Hello World
```

Lets add few more phony targets - `generate` and `clean` to the `Makefile`:

```
say_hello:
    @echo "Hello World"

generate:
    @echo "Creating empty text files..."
    touch file-{1..10}.txt

clean:
    @echo "Cleaning up..."
    rm *.txt
```

If we try to run `make` after the changes, only the target `say_hello` will be executed run. That's because only the first target in the makefile is the default target. Often called as *default goal*. This is the reason you will see `all` as the first target in most projects. It is the responsibility of `all` to call other targets. We can override this behavior using a special phony target called `.DEFAULT_GOAL`. Lets include that at the beginning of our makefile:

```
.DEFAULT_GOAL := generate
```

This will ensure to run the target `generate` as default:

```
$ make
Creating empty text files...
touch file-{1..10}.txt
```

As the name suggest the phony target `.DEFAULT_GOAL` can only run one target at a time. This is also the reason most makefiles includes `all` as a target which can call as many targets as needed. Lets include the phony target `all` and remove `.DEFAULT_GOAL`:

```
all: say_hello generate

say_hello:
```

```
    @echo "Hello World"

generate:
    @echo "Creating empty text files..."
    touch file-{1..10}.txt

clean:
    @echo "Cleaning up..."
    rm *.txt
```

Before running `make` lets include another special phony target `.PHONY` where we define all the target which are not files. `make` will run its recipe un-conditionally, regardless of whether a file with that name exists or what its last-modification time is. Here is the complete makefile:

```
.PHONY: all say_hello generate clean

all: say_hello generate

say_hello:
    @echo "Hello World"

generate:
    @echo "Creating empty text files..."
    touch file-{1..10}.txt

clean:
    @echo "Cleaning up..."
    rm *.txt
```

The `make` should call `say_hello` and `generate`:

```
$ make
Hello World
Creating empty text files...
touch file-{1..10}.txt
```

It is a good practice not to call `clean` in `all` or put it as the first target. `clean` should be called manually when cleaning is needed as a first argument to `make`:

```
$ make clean
Cleaning up...
rm *.txt
```

Now you got an idea how a basic makefile works and how to write a simple makefile. Lets see some advance usage of makefile.

Advance examples

Variables

In above example most of target and prerequisite values are hard coded but in real life projects those are replaces with variables and patterns.

A simplest way to define a variable in a makefile is using `=` operator. For example to assign the command `gcc` to a variable `CC`:

```
CC = gcc
```

This is also called *recursive expanded variable* and used in a rule as below:

```
hello: hello.c
    ${CC} hello.c -o hello
```

As you might have guessed, the recipe expands as below when it is passed to the terminal:

```
gcc hello.c -o hello
```

Both `${CC}` and `$(CC)` are valid references to call `gcc`. But if one tries to reassign a variable to itself, it will cause an infinite loop. Lets verify this:

```
CC = gcc
CC = ${CC}

all:
    @echo ${CC}
```

Running `make` will result in:

```
$ make
Makefile:8: *** Recursive variable 'CC' references itself (eventually).  Stop.
```

To avoid this kind of scenario, we can use `:=` operator. Also called as *simply expanded variable*. We should have no problem running below makefile:

```
CC := gcc
CC := ${CC}

all:
    @echo ${CC}
```

Patterns and Functions

Below makefile can compile all C programs by using variables, patterns and functions. Lets explore it line by line:

```
# Usage:
# make        # compile all binary
# make clean  # remove ALL binaries and objects

.PHONY = all clean

CC = gcc            # compiler to use

LINKERFLAG = -lm

SRCS := $(wildcard *.c)
BINS := $(SRCS:%.c=%)

all: ${BINS}

%: %.o
    @echo "Checking.."
    ${CC} ${LINKERFLAG} $< -o $@

%.o: %.c
    @echo "Creating object.."
    ${CC} -c $<

clean:
    @echo "Cleaning up..."
    rm -rvf *.o ${BINS}
```

- Lines starting with `#` are comments.
- Line `.PHONY = all clean` defines phony targets `all` and `clean`.
- Variable `LINKERFLAG` defines flags to be used with `gcc` in a recipe.
- `SRCS := $(wildcard *.c)`: `$(wildcard pattern)` is one of the *function for filenames*. In this case all files having `.c` extension will be stored in a variable `SRCS`.
- `BINS := $(SRCS:%.c=%)`: This is called as *substitution reference*. In this case if `SRCS` has values `'foo.c bar.c'`, `BINS` will have `'foo bar'`.
- Line `all: ${BINS}`: The phony target `all` calls values in`${BINS}` as individual targets.
- Rule:

  ```
  %: %.o
      @echo "Checking.."
      ${CC} ${LINKERFLAG} $< -o $@
  ```

  Lets take an example to understand this rule. Suppose `foo` is one of the

value in `${BINS}` then `%` will match `foo`(`%` can match any target name) Below is the rule in its expanded form:

```
foo: foo.o
    @echo "Checking.."
    gcc -lm foo.o -o foo
```

As seen `%` is replaced by `foo`. `$<` is replaced by `foo.o`. `$<` is pattern to match prerequisite and `$@` matches the target. This rule will be called for every value in `${BINS}`

- Rule:

```
%.o: %.c
    @echo "Creating object.."
    ${CC} -c $<
```

Every prerequisite in previous rule considered as a target for this rule. Below is the rule in its expanded form:

```
foo.o: foo.c
    @echo "Creating object.."
    gcc -c foo.c
```

- Finally we remove all binaries and object files in target `clean`

Below is re-write of above makefile assuming it is placed in the directory having a single file `foo.c`

```
# Usage:
# make        # compile all binary
# make clean  # remove ALL binaries and objects

.PHONY = all clean

CC = gcc            # compiler to use

LINKERFLAG = -lm

SRCS := foo.c
BINS := foo

all: foo

foo: foo.o
    @echo "Checking.."
    gcc -lm foo.o -o foo

foo.o: foo.c
    @echo "Creating object.."
```

```
    ${CC} -c foo.c

clean:
    @echo "Cleaning up..."
    rm -rvf foo.o foo
```

There is lot more to explore apart from what we covered in this post. Please refer the GNU Make manual for complete reference and examples.