# What is a Makefile and how does it work?

Run and compile your programs more efficiently with this handy automation tool.

By Sachin Patil

August 22, 2018 | 8 Comments | 7 min read

638 readers like this.

***Image by:*** *Opensource.com*

If you want to run or update a task when certain files are updated, the `make` utility can come in 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 open source projects use `make` to compile a final executable binary, which can then be installed using `make install`.

In this article, we'll explore `make` and `Makefile` using basic and advanced examples. Before you start, ensure that `make` is installed in your system.

## Basic examples

Let's start by printing the classic "Hello World" on the terminal. Create a empty directory `myproject` containing a file `Makefile` with this content:

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

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

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

In the example above, `say_hello` behaves like a function name, as in any programming language. This is called the *target*. The *prerequisites* or *dependencies* follow the target. For the sake of simplicity, we have not defined any prerequisites in this example. The command `echo "Hello World"` is called the *recipe*. The *recipe* uses *prerequisites* to make a *target*. The target, prerequisites, and recipes together make 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 that depends on prerequisites (source files). On the other hand, a prerequisite can also be a target that depends on other dependencies:

```
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 for the target to be a file; it could be just a name for the recipe, as in our example. We call these "phony targets."

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

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

Now try to run `make` again. The output should display only this:

```
$ make
Hello World
```

Let's add a 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. That's because only the first target in the makefile is the default target. Often called the *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`.

Let's include that at the beginning of our makefile:

```
.DEFAULT_GOAL := generate
```

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

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

As the name suggests, the phony target `.DEFAULT_GOAL` can run only one target at a time. This is why most makefiles include `all` as a target that can call as many targets as needed.

Let's 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`, let's include another special phony target, `.PHONY`, where we define all the targets that are not files. `make` will run its recipe 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 that you have an idea of how a basic makefile works and how to write a simple makefile, let's look at some more advanced examples.

## Advanced examples

### Variables

## More Linux resources

**Linux commands cheat sheet**

**Advanced Linux commands cheat sheet**

**Free online course: RHEL Technical Overview**

**Linux networking cheat sheet**

**SELinux cheat sheet**

**Linux common commands cheat sheet**

**What are Linux containers?**

**Our latest Linux articles**

In the above example, most target and prerequisite values are hard-coded, but in real projects, these are replaced with variables and patterns.

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

```
CC = gcc
```

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

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

As you may 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. Let's verify this:

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

all:
```

```
    @echo ${CC}
```

Running `make` will result in:

```
$ make
Makefile:8: *** Recursive variable 'CC' references itself (e
```

To avoid this scenario, we can use the `:=` operator (this is also called the *simply expanded variable*). We should have no problem running the makefile below:

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

all:
    @echo ${CC}
```

### Patterns and functions

The following makefile can compile all C programs by using variables, patterns, and functions. Let's 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 *functions for filenames*. In this case, all files with the `.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} $&lt; -o $@
```

Let's look at an example to understand this rule. Suppose `foo` is one of the values 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 shown, `%` is replaced by `foo`. `$<` is replaced by `foo.o`. `$<` is patterned to match prerequisites and `$@` matches the target. This rule will be called for every value in `${BINS}`

Rule:

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

Every prerequisite in the previous rule is considered 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 the rewrite of the above makefile, assuming it is placed in the directory having a single file `foo.c`:

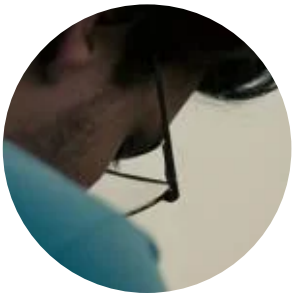
```
# Usage:
# make        # compile all binary
# make clean  # remove ALL binaries and objects

.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.."
        gcc -c foo.c

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

For more on makefiles, refer to the GNU Make manual, which offers a complete reference and examples.

You can also read our Introduction to GNU Autotools to learn how to automate the generation of a makefile for your coding project.

**Tags:** LINUX TOOLS




## Sachin Patil

Sachin is passionate about Free and Open source software. He is avid GNU Emacs user and likes to talk and write about open source, GNU/Linux, Git, and Python. He has previously worked on OpenStack, ManageIQ/CloudForms & Red Hat Insights. He also likes to explore Swift Object Storage in his spare time. He can be reached on IRC as psachin@{Libera.Chat, Freenode,

OFTC, gnome}.

**More about me**

## 8 Comments

These comments are closed.

Kiko Fernandez-Reyes | August 22, 2018

No readers like this yet.

Hi,

Thanks for this tutorial. Really useful and easy to follow.

Sachin Patil | August 22, 2018

No readers like this yet.

Glad that you liked it. Thanks!!

Thomas | August 23, 2018

No readers like this yet.

Interesting article. It motivate me to play around with Makefiles.

Thank you

Sachin Patil | August 24, 2018

No readers like this yet.

Cool!!

RogersGuedes | September 11, 2018

No readers like this yet.

Thank you so much!

Sachin Patil | November 21, 2018

No readers like this yet.

:)

Neeraj Sharma | November 5, 2018

No readers like this yet.

Thank you so much...
very nice article...

Sachin Patil | November 15, 2018

No readers like this yet.

Thanks!

## Related Content

What's new in GNOME 44?

5 reasons virtual machines still matter

Remove the background from an image with this Linux command

## ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

**Opensource.com** aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.