FREE eBook

LEARNING makefile

Free unaffiliated eBook created from
Stack Overflow contributors.

#makefile

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: makefile

It is an unofficial and free makefile ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official makefile.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with makefile

Remarks

A *makefile* is a text file which controls the operation of the `make` program. The `make` program is typically used to manage the creation of programs from their source files, but it can be more generally used to handle any process where files (or *targets*) need to be regenerated after other files (or *prerequisites*) have been modified. The *makefile* describes the relationship between targets and prerequisites, and also specifies the commands needed to bring the target up-to-date when one or more of the prerequisites has changed. The only way that `make` determines "out of date-ness" is by comparing the modification time of target files and their prerequisites.

Makefiles are somewhat unique in a few ways which can be confusing initially.

First, a makefile consists of two completely different programming languages in the same file. The bulk of the file is written in a language that `make` can understand: this provides variable assignment and expansion, some preprocessor capabilities (including other files, conditional parsing of sections of the file, etc.) as well as the definition of targets and their prerequisites. In addition, each target can have a *recipe* associated with it which specifies which commands should be invoked to cause that target to be brought up-to-date. The recipe is written as a shell script (POSIX sh by default). The `make` program doesn't parse this script: it runs a shell and passes the script to the shell to be run. The fact that recipes are not parsed by `make`, but instead handled by a separate shell process, is central in understanding makefiles.

Second, a makefile is not a procedural language like a script: as `make` parses the makefile it constructs a *directed graph* internally where targets are the nodes of the graph and the prerequisite relationships are the edges. Only after all makefiles have been completely parsed and the graph is complete will `make` choose one node (target) and attempt to bring it up to date. In order to ensure a target is up to date, it must first ensure that each of that target's prerequisites are up to date, and so on recursively.

Versions

Name	Also Known As	Initial Version	Version	Release Date
POSIX make		1992	IEEE Std 1003.1-2008, 2016 Edition	2016-09-30
NetBSD make	bmake	1988	20160926	2016-09-26
GNU make	gmake	1988	4.2.1	2016-06-10
SunPro make	dmake	2006		2015-07-13

Name	Also Known As	Initial Version	Version	Release Date
MSVS nmake		2003	2015p3	2016-06-27

Examples

Basic Makefile

Consider writing a "hello world!" program in c. Lets say our source code is in a file called source.c, now in order to run our program we need to compile it, typically on Linux (using gcc) we would need to type `$> gcc source.c -o output` where output is the name of the executable to be generated. For a basic program this works well but as programs become more complex our compilation command can also become more complex. This is where a *Makefile* comes in, makefiles allow us to write out a fairly complex set of rules for how to compile a program and then simply compile it by typing make on the command line. For instance here is a possible example Makefile for the hello wold example above.

Basic Makefile

Lets make a basic *Makefile* and save it to our system in the same directory as our source code named *Makefile*. Note that this file needs to be named Makefile, however the capitol M is optional. That said it is relatively standard to use a capitol M.

```
output: source.c
    gcc source.c -o output
```

Note that there is exactly one tab before the gcc command on the second line (this is important in makefiles). Once this Makefile is written every time the user types make (in the same directory as the Makefile) make will check to see if source.c has been modified (checks the time stamp) if it has been modified more recently than output it will run the compilation rule on the following line.

Variables in Makefiles

Depending on the project you may want to introduce some variables to your make file. Here is an example Makefile with variables present.

```
CFLAGS = -g -Wall

output: source.c
    gcc $< $(CFLAGS) -o $@
```

Now lets explore what happened here. In the first line we declared a variable named CFLAGS that holds several common flags you may wish to pass to the compiler, note that you can store as many flags as you like in this variable. Then we have the same line as before telling make to check source.c to see if it has been changed more recently than output, if so it runs the compilation rule. Our compilation rule is mostly the same as before but it has been shortened by

using variables, the `$<` variable is built into make (referred to as an automatic variable see *https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html*) and it always stands for the source so in this case *source.c*. `$(CFLAGS)` is our variable that we defined before, but note that we had to put the variable in parenthesis with a $ in front like this`$(someVariable)`. This is the syntax for telling Make to expand the variable out to what you typed before. Finally we have the $@ symbol, once again this is a variable built into make, and it simply stands for the target of the compilation step, so in this case it stands for *output*.

Clean

Make clean is another useful concept to learn about make files. Lets modify the *Makefile* from above

```
CFLAGS = -g -Wall
TARGETS = output

output: source.c
    gcc $< $(CFLAGS) -o $@

clean:
    rm $(TARGETS)
```

As you can see we simply added one more rule to our *Makefile*, and one additional variable that contains all of our targets. This is a somewhat common rule to have in makefiles as it allows you to quickly remove all of the binaries you produced simply by typing `$> make clean`. By typing make clean you tell the make program to run the clean rule and then make will run the rm command to delete all of your targets.

I hope this brief overview of using make helps you speed up your workflow, *Makefiles* can become very complex, but with these ideas you should be able to get started using make and have a better understanding of what is going on in other programmers *Makefiles*. For more information about using make an excellent resource is https://www.gnu.org/software/make/manual/.

Defining Rules

Quick start

A rule describes when and how certain files (rule's **targets**) are created. It can also serve to update a target file if any of the files required for its creation (target's **prerequisites**) are newer than the target.

Rules follow the syntax below: (Note that *commands* following a rule are indented by a **tab**)

```
targets: prerequisites
        <commands>
```

where *targets* and *prerequisites* are file names or special reserved names and *commands* (if present) are executed by a shell to build/rebuild *targets* that are out-of-date.

To execute a rule one can simply run the `make` command in the terminal from the same directory where the *Makefile* resides. Running `make` without specifying the target, will execute the first rule defined in the *Makefile*. By convention, the first rule in the *Makefile* is often called *all* or *default*, commonly listing all valid build targets as prerequisites.

`make` only executes the rule if the target is out-of-date, meaning either it doesn't exist or its modification time is older than any of its prerequisites. If the list of prerequisites is empty, the rule will only be executed when it is first invoked to build the targets. However, when the rule does not create a file and the target is a dummy variable, the rule will always be executed.

GNU make

Pattern Rules

Pattern rules are used to specify multiple targets and construct prerequisite names from target names. They are more general and more powerful compared to ordinary rules as each target can have its own prerequisites. In pattern rules, a relationship between a target and a prerequisite is build based on prefixes including path names and suffixes, or both.

Imagine we want to build the targets `foo.o` and `bar.o`, by compiling C scripts, `foo.c` and `bar.c`, respectively. This could be done by using the ordinary rules below:

```
foo.o: foo.c
    cc -c $< -o $@

bar.o: bar.c
    cc -c $< -o $@
```

where *automatic variable* `$<` is the name of the first prerequisite and `$@` the name of the target (A complete list of automatic variables can be found here).

However, as the targets share the same suffix, the above two rules can now be substituted by the following pattern rule:

```
%.o: %.c
    cc -c $< -o $@
```

Implicit Rules

Implicit rules tell `make` how to use customary methods to build certain types of target files, which are used very often. `make` uses the target file name to determine which implicit rule to invoke.

The pattern rule example we saw in the previous section, does not actually need to be declared in a *Makefile* as `make` has an implicit rule for C compilation. Thus, in the following rule, the prerequisites `foo.o` and `bar.o` will be build using the implicit rule for C compilation, before building `foo`.

```
foo : foo.o bar.o
    cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

A catalogue of implicit rules and the variables used by them can be found here.

generic rule to gzip a file

if a directory contain 2 files:

```
$ ls
makefile
example.txt
```

and `makefile` contain the following text

```
%.gz: %
    gzip $<
```

then you can obtain `example.txt.gz` by typing in the shell

```
$ make -f makefile example.txt.gz
```

the makefile consist of only one rule that instruct *make* how to create a file whose name end with *.gz* if there is a file with the same name but the *.gz* suffix.

makefile Hello World

C:\makefile:

```
helloWorld :
[TAB]echo hello world
```

run results:

```
C:\>make
echo hello world
hello world
```

Note: [TAB] should be replaced by an actual tab, stackoverflow replaces tabs with spaces, and spaces are not used the same as tabs in a makefile.

Read Getting started with makefile online: https://riptutorial.com/makefile/topic/1793/getting-started-with-makefile

Chapter 2: .PHONY target

Examples

Using .PHONY for non-files targets

Use `.PHONY` to specify the targets that are not files, e.g., `clean` or `mrproper`.

Good example

```
.PHONY: clean
clean:
    rm *.o temp
```

Bad example

```
clean:
    rm *.o temp
```

In the good example `make` knows that `clean` is not a file, therefore it will not search if it is or not up to date and will execute the recipe.

In the bad example `make` will look for a file named `clean`. If it doesn't exist or is not up to date it will execute the recipe, but if it does exist and is up to date the recipe will not be executed.

Using .PHONY for recursive invocations of 'make' command

Recursive use of make means using make as a command within a makefile. This technique is useful when a large project contains sub-directories, each having their respective makefiles. The following example will help understand advantage of using .PHONY with recursive make.

```
/main
     |_ Makefile
     |_ /foo
            |_ Makefile
            |_ ... // other files
     |_ /bar
            |_ Makefile
            |_ ... // other files
     |_ /koo
            |_ Makefile
            |_ ... // other files
```

To run sub-directory's makefile from within the makefile of main, the main's makefile would have looping as shown below (there are other ways in which this can be achieved, but that is out of scope of the current topic)

```
SUBDIRS = foo bar koo
```

```
subdirs:
        for dir in $(SUBDIRS); do \
          $(MAKE) -C $$dir; \
        done
```

However, there are pitfalls with this method.

1. Any error detected in a sub-make is ignored by this rule, so it will continue to build the rest of the directories even when one fails.
2. Make's ability to perform Parallel execution of multiple build targets is not utilized since only one rule is used.

By declaring the sub-directories as .PHONY targets (you must do this as the sub-directory obviously always exists; otherwise it won't be built) these problems can be overcome.

```
SUBDIRS = foo bar koo

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
        $(MAKE) -C $@
```

Read .PHONY target online: https://riptutorial.com/makefile/topic/5542/-phony-target

Chapter 3: Advanced Makefile

Examples

Building from different source folders to different target folders

Main features of this Makefile :

- Automatic detection of C sources in specified folders
- Multiple source folders
- Multiple corresponding target folders for object and dependency files
- Automatic rule generation for each target folder
- Creation of target folders when they don't exist
- Dependency management with `gcc` : Build only what is necessary
- Works on `Unix` and `DOS` systems
- Written for GNU Make

This Makefile can be used to build a project with this kind of structure :

```
\---Project
    +---Sources
    |   +---Folder0
    |   |       main.c
    |   |
    |   +---Folder1
    |   |       file1_1.c
    |   |       file1_1.h
    |   |
    |   \---Folder2
    |           file2_1.c
    |           file2_1.h
    |           file2_2.c
    |           file2_2.h
    \---Build
        |   Makefile
        |   myApp.exe
        |
        +---Folder0
        |       main.d
        |       main.o
        |
        +---Folder1
        |       file1_1.d
        |       file1_1.o
        |
        \---Folder2
                file2_1.d
                file2_1.o
                file2_2.d
                file2_2.o
```

Makefile

```
# Set project directory one level above of Makefile directory. $(CURDIR) is a GNU make
variable containing the path to the current working directory
PROJDIR := $(realpath $(CURDIR)/..)
SOURCEDIR := $(PROJDIR)/Sources
BUILDDIR := $(PROJDIR)/Build

# Name of the final executable
TARGET = myApp.exe

# Decide whether the commands will be shwon or not
VERBOSE = TRUE

# Create the list of directories
DIRS = Folder0 Folder1 Folder2
SOURCEDIRS = $(foreach dir, $(DIRS), $(addprefix $(SOURCEDIR)/, $(dir)))
TARGETDIRS = $(foreach dir, $(DIRS), $(addprefix $(BUILDDIR)/, $(dir)))

# Generate the GCC includes parameters by adding -I before each source folder
INCLUDES = $(foreach dir, $(SOURCEDIRS), $(addprefix -I, $(dir)))

# Add this list to VPATH, the place make will look for the source files
VPATH = $(SOURCEDIRS)

# Create a list of *.c sources in DIRS
SOURCES = $(foreach dir,$(SOURCEDIRS),$(wildcard $(dir)/*.c))

# Define objects for all sources
OBJS := $(subst $(SOURCEDIR),$(BUILDDIR),$(SOURCES:.c=.o))

# Define dependencies files for all objects
DEPS = $(OBJS:.o=.d)

# Name the compiler
CC = gcc

# OS specific part
ifeq ($(OS),Windows_NT)
    RM = del /F /Q
    RMDIR = -RMDIR /S /Q
    MKDIR = -mkdir
    ERRIGNORE = 2>NUL || true
    SEP=\\
else
    RM = rm -rf
    RMDIR = rm -rf
    MKDIR = mkdir -p
    ERRIGNORE = 2>/dev/null
    SEP=/
endif

# Remove space after separator
PSEP = $(strip $(SEP))

# Hide or not the calls depending of VERBOSE
ifeq ($(VERBOSE),TRUE)
    HIDE =
else
    HIDE = @
endif

# Define the function that will generate each rule
```

```
define generateRules
$(1)/%.o: %.c
    @echo Building $$@
    $(HIDE)$(CC) -c $$(INCLUDES) -o $$(subst /,$$(PSEP),$$@) $$(subst /,$$(PSEP),$$<) -MMD
endef

.PHONY: all clean directories

all: directories $(TARGET)

$(TARGET): $(OBJS)
    $(HIDE)echo Linking $@
    $(HIDE)$(CC) $(OBJS) -o $(TARGET)

# Include dependencies
-include $(DEPS)

# Generate rules
$(foreach targetdir, $(TARGETDIRS), $(eval $(call generateRules, $(targetdir))))

directories:
    $(HIDE)$(MKDIR) $(subst /,$(PSEP),$(TARGETDIRS)) $(ERRIGNORE)

# Remove all objects, dependencies and executable files generated during the build
clean:
    $(HIDE)$(RMDIR) $(subst /,$(PSEP),$(TARGETDIRS)) $(ERRIGNORE)
    $(HIDE)$(RM) $(TARGET) $(ERRIGNORE)
    @echo Cleaning done !
```

How to use this Makefile To adapt this Makefile to your project you have to :

1. Change the `TARGET` variable to match your target name
2. Change the name of the `Sources` and `Build` folders in `SOURCEDIR` and `BUILDDIR`
3. Change the verbosity level of the Makefile in the Makefile itself or in make call
4. Change the name of the folders in `DIRS` to match your sources and build folders
5. If required, change the compiler and the flags

Zipping lists

GNU make

This `pairmap` function takes three arguments:

1. A function name
2. First space-separated list
3. Second space-separated list

For each zipped tuple in the lists it will call the function with the following arguments:

1. Tuple element from the first list
2. Tuple element from the second list

It will expand to a space-separated list of the function expansions.

```
list-rem = $(wordlist 2,$(words $1),$1)
pairmap = $(and $(strip $2),$(strip $3),$(call \
    $1,$(firstword $2),$(firstword $3)) $(call \
    pairmap,$1,$(call list-rem,$2),$(call list-rem,$3)))
```

For example, this:

```
LIST1 := foo bar baz
LIST2 := 1 2 3

func = $1-$2

all:
    @echo $(call pairmap,func,$(LIST1),$(LIST2))

.PHONY: all
```

Will print `foo-1 bar-2 baz-3`.

Read Advanced Makefile online: https://riptutorial.com/makefile/topic/6154/advanced-makefile

Chapter 4: GNU Pattern Rules

Examples

Basic Pattern Rule

A pattern rule is indicated by a single `%` character in the target. The `%` matches a non-empty string called the **stem**. The stem is then substituted for every `%` that appears in the prerequisite list.

For example, this rule:

```
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

Will match any target ending in `.o`. If the target was `foo.o`, the stem would be `foo` and it would compile `foo.c` to `foo.o`. Targets and prerequisites can be accessed using automatic variables.

Targets matching multiple Pattern Rules

If a target matches multiple pattern rules, make will use the one whose prerequisites exist or can be built. For example:

```
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
%.o: %.s
    $(AS) $(ASFLAGS) $< -o $@
```

Will compile `foo.c` to `foo.o` or assemble `foo.s` to `foo.o`, depending on which one of `foo.c` or `foo.s` exists.

If multiple rules have prerequisites that exist or can be built, make will use the rule that matches to the shortest stem. For example:

```
f%r:
    @echo Stem is: $*
fo%r:
    @echo Stem is: $*
```

Will use the second rule to make the target `foo.bar`, echoing `Stem is: o.ba`.

If multiple rules match to the shortest stem, make will use the first one in the Makefile.

Directories in Pattern Rules

If the target pattern doesn't contain slashes, make will remove the directory part from the target it's trying to build before matching. The directory will then be put in front of the stem. When the stem is used to build the target name and prerequisites, the directory part is stripped from it, the stem is

substituted in place of the `%` and finally the directory is put in front of the string. For example:

```
foo%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

Will match `lib/foobar.o`, with:

- Stem (`$*`): `lib/bar`
- Target name (`$@`): `lib/foobar.o`
- Prerequisites (`$<`, `$^`): `lib/foobar.c`

In this example, a `lib/foo%.o` rule would take precedence over the `foo%.o` rule because it matches to a shorter stem.

Pattern Rules with multiple targets

Pattern rules can have multiple targets but, unlike normal rules, the recipe is responsible for making all the targets. For example:

```
debug/%.o release/%.o: %.c
    $(CC) $(CFLAGS_DEBUG) -c $< -o debug/$*.o
    $(CC) $(CFLAGS_RELEASE) -c $< -o release/$*.o
```

Is a valid rule, which will build both debug and release objects when one of them has to be built. If we wrote something like:

```
debug/%.o release/%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

It would work when only one of `debug/*.o` or `release/*.o` is built, but it would only build the first target (and consider the second one to be up-to-date) when both have to be built.

Read GNU Pattern Rules online: https://riptutorial.com/makefile/topic/6860/gnu-pattern-rules

Chapter 5: Variables

Examples

Referencing a Variable

To use the value stored in a variable, use the dollar sign followed by the variable name enclosed by parentheses or curly braces.

```
x = hello
y = $(x)
# y now contains the value "hello"
y = ${x}
# parentheses and curly braces are treated exactly the same
```

If a variable's name is only one character long, the parentheses/braces can be omitted (e.g., `$x`). This practice is used for automatic variables (see below), but is not recommended for general-purpose variables.

Simply-Expanded Variables

Simply-expanded variables behave like variables from traditional programming languages. The expression on the right-hand side is evaluated, and the result is stored in the variable. If the right-hand side contains a variable reference, that variable is expanded before the assignment takes place.

```
x := hello
y := $(x)
# Both $(x) and $(y) will now yield "hello"
x := world
# $(x) will now yield "world", and $(y) will yield "hello"
```

An alternative form is to use double-colon assignment:

```
x ::= hello
```

Single- and double-colon assignment are equivalent. The POSIX make standard only mentions the `::=` form, so implementations with strict standards compliance may not support the single-colon version.

Recursively-Expanded Variables

When defining a recursively-expanded variable, the contents of the right-hand side are stored as-is. If a variable reference is present, the reference itself is stored (not the value of the variable). Make waits to expand the variable references until the variable is actually used.

```
x = hello
```

```
y = $(x)
# Both $(x) and $(y) will now yield "hello"
x = world
# Both $(x) and $(y) will now yield "world"
```

In this example, the definition of `y` is recursive. The reference to `$(x)` doesn't get expanded until `$(y)` is expanded. This means that whenever the value of `x` changes, the value of `y` will change as well.

Recursively-expanded variables are a powerful but easily-misunderstood tool. They can be used to create constructs that resemble templates or functions, or even to automatically generate portions of a makefile. They can also be the source of hard-to-debug problems. Be careful to only use recursively-expanded variables when necessary.

Automatic Variables

Within the context of an individual rule, Make automatically defines a number of special variables. These variables can have a different value for each rule in a makefile and are designed to make writing rules simpler. These variables can only be used in the recipe portion of a rule.

Variable	Description
`$@`	File name of the rule's target
`$%`	The target member's name, if the rule's target is an archive
`$<`	File name of the first prerequisite
`$^`	List of all prerequisites
`$?`	List of all prerequisites that are newer than the target
`$*`	The "stem" of an implicit or pattern rule

The following example uses automatic variables to generate a generic rule. This instructs make how to construct a .o file out of a .c file with the same name. Since we don't know the specific name of the affected files, we use `$@` as a placeholder for the output file's name and `$^` as a placeholder for the prerequisite list (in this case, the list of input files).

```
%.o: %.c
    cc -Wall $^ -c $@
```

Conditional Variable Assignment

The `?=` operator is an extension that behaves like `=`, except that the assignment *only* occurs if the variable is not already set.

```
x = hello
```

```
x ?= world
# $(x) will yield "hello"
```

Appending Text To an Existing Variable

The `+=` operator is a common extension that adds the specified content to the end of the variable, separated by a space.

```
x = hello
x += world
```

Variable references in the right-hand side will be expanded if and only if the original variable was defined as a simply-expanded variable.

Read Variables online: https://riptutorial.com/makefile/topic/6191/variables

Credits

S. No	Chapters	Contributors
1	Getting started with makefile	chen, Community, Dan , Eldar Abusalimov, kdhp, levif, MadScientist, mox, mxenoph, SketchBookGames
2	.PHONY target	kdhp, madD7, TimF
3	Advanced Makefile	Andrea Biondo, TimF
4	GNU Pattern Rules	Andrea Biondo, kdhp
5	Variables	bta