Makefile Cheatsheet

A comprehensive cheat sheet for Makefiles, covering syntax, variables, rules, functions, and command prefixes, along with practical examples.

Makefile Basics

Syntax Overview

A Makefile consists of rules, variables, and directives.

General Structure:

```
target: prerequisites
    command
```

- `target`: The file to be created or updated.
- `prerequisites`: Files required for the target.
- `command`: Action to be executed.

Comments:

```
# This is a comment
```

Including Makefiles:

```
include other.mk
-include optional.mk # Ignore if it doesn't exist
```

Variables

Variable Assignment

```
VAR = value         # Recursive assignment
VAR := value        # Simple assignment
VAR ?= value        # Conditional assignment
VAR += more_value   # Append
```

Variable Usage

```
$(VAR)              # Access variable
${VAR}              # Alternative syntax
```

Example

```
SRC = main.c utils.c
OBJ = $(SRC:.c=.o)  # Substitutes .c with .o
all: $(OBJ)
    gcc -o myprogram $(OBJ)
```

Rules

Explicit Rule

```
target: prerequisite1 prerequisite2
    command1
    command2
```

Implicit Rule

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

Pattern Rule

```
$(OBJ): %.o: %.c
    gcc -c -o $@ $<
```

Advanced Features

Functions

String Functions	`$(subst FROM,TO,TEXT)`
	`# Substitution`
	`$(patsubst PATTERN,REPLACEMENT,TEXT)  # Pattern substitution`
	`$(strip STRING)`
	`# Remove leading/trailing whitespace`
	`$(findstring FIND,IN)`
	`# Find string`
	`$(filter PATTERN,TEXT)`
	`# Filter matching words`
	`$(filter-out PATTERN,TEXT)`
	`# Filter out matching words`
	`$(sort LIST)`
	`# Sort list`
	`$(word N,TEXT)`
	`# Extract nth word`
	`$(wordlist START,END,TEXT)`
	`# Extract wordlist`
	`$(words TEXT)`
	`# Count words`
	`$(firstword TEXT)`
	`# First word`

File Name Functions	`$(dir NAMES)  # Directory part`
	`$(notdir NAMES)  # Non-directory part`
	`$(suffix NAMES)  # Suffix part`
	`$(basename NAMES)  # Basename part`
	`$(addsuffix SUFFIX,NAMES)  # Add suffix`
	`$(addprefix PREFIX,NAMES)  # Add prefix`
	`$(join LIST1,LIST2)  # Join lists`
	`$(wildcard PATTERN)  # Wildcard expansion`
	`$(realpath NAMES)  # Canonicalize file names`
	`$(abspath NAMES)  # Absolute file name`

Conditional Functions	`$(if CONDITION,THEN-PART,ELSE-PART)`
	`$(or CONDITION1,CONDITION2,...)`
	`$(and CONDITION1,CONDITION2,...)`

Directives

Conditional Directives	`ifeq (ARG1, ARG2)`
	`  ...commands...`
	`else`
	`  ...commands...`
	`endif`
	
	`ifdef VARIABLE`
	`  ...commands...`
	`else`
	`  ...commands...`
	`endif`

Include Directive	`include filenames...` `-include filenames... # Non-fatal`
Override Directive	`variable := value` `override variable := new_value`

Command Execution

Commands are executed by the shell. Each command is executed in a separate subshell.

Example:

```
all:
    echo "Starting..."
    date
    echo "Done."
```

Use `$(shell command)` to execute a shell command and use its output as a variable value.

Example:

```
VERSION := $(shell git describe --tags --abbrev=0)
```

Common Patterns & Best Practices

Target-Specific Variable Values

You can define variable values that are specific to a target.

Syntax:

```
target : variable = value
```

Example:

```
foo.o : CFLAGS = -O2
bar.o : CFLAGS = -g
```

Pattern-Specific Variable Values

You can define variable values that are specific to a pattern of targets.

Syntax:

```
%.o : CFLAGS = -O2
```

This sets `CFLAGS` to `-O2` for all `.o` files.

Order-only Prerequisites

Order-only prerequisites are listed after a pipe symbol `|`. They ensure that certain targets are built before the current target, but they don't cause the current target to rebuild if they are updated.

Syntax:

```
target: normal-prerequisites | order-only-prerequisites
```

Example:

```
all: myprogram

myprogram: foo.o bar.o | config.h
    gcc -o myprogram foo.o bar.o

config.h:
    ./configure
```

Phony Targets

Phony targets are targets that do not represent actual files. They are typically used to define actions like `clean`, `all`, `install`, etc.

Syntax:

```
.PHONY: target_name
```

Example:

```
.PHONY: all clean

all: myprogram

clean:
    rm -f *.o myprogram
```

Debugging and Options

Makefile Options

`make`	Starts make process.
`make -f <filename>`	Specifies the makefile to use.
`make -n` or `make --just-print`	Prints the commands that would be executed, without actually executing them (dry run).
`make -B` or `make --always-make`	Unconditionally make all targets.
`make -j [N]` or `make --jobs=[N]`	Specifies the number of jobs to run simultaneously. If N is omitted, make runs as many jobs simultaneously as possible.
`make -k` or `make --keep-going`	Continue as much as possible after an error.

Debugging Tips

Use `make -n` or `make --just-print` to see the commands that Make will execute.
Use `make -d` for verbose output, including variable assignments and implicit rules.
Use `$(warning TEXT)` or `$(error TEXT)` to print debugging messages during Makefile parsing.

Example Makefile

```
# Variables
CC = gcc
CFLAGS = -Wall -g
SRC = main.c helper.c
OBJ = $(SRC:.c=.o)
TARGET = myapp

# Phony target
.PHONY: all clean

# Default target
all: $(TARGET)

# Link the object files to create the target
$(TARGET): $(OBJ)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJ)

# Compile C source files to object files
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

# Clean target
clean:
    rm -f $(OBJ) $(TARGET)
```