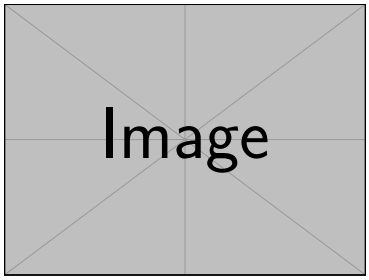
Complete Makefile Guide

From Basics to Advanced Features



Compiled: December 25, 2025

Version 1.0

Contents

1 Introduction to Make

Make is a build automation tool that automatically builds executable programs and libraries from source code by reading files called **Makefiles** which specify how to derive the target program.

Information

Key Points:

- Make is a scripting tool used for building executables according to commands
- It parses an input file called "Makefile" that defines rules to execute
- There are different implementations: GNU Make, Borland Make, etc.

1.1 Why We Need Make?

Make is essential for several reasons:

- **Organize compilation process**: Manage complex build dependencies
- **Automate repetitive tasks**: Run tools and scripts automatically
- **Save time**: Only rebuild what's necessary (incremental builds)
- **Full control**: Manage outputs and their timestamps

1.2 Basic Syntax

Rule Definition

The basic syntax of a Make rule is:

```
target: prerequisites
    recipe
```

Where:

- **target**: File to be created or action to perform
- **prerequisites**: Files needed to create the target
- **recipe**: Commands to execute (must start with TAB)

Warning

Important: Recipe lines must start with a TAB character, not spaces!

Variables

Variables in Makefiles:

```
# Define variable
files = main.cpp
# Append to variable
files += hello.cpp
```

```
# Use variable
$(files)
```

Special Features

```
# Disable echo (silent execution)
@echo "This won't show the command"

# Access environment variables
echo $(PATH)

# Execute shell commands
clean:
    @rm *.o
    @echo "`date` done" > log.txt

# Run external scripts
test:
    python analysis.py
```

2 Functions in Makefiles

2.1 Defining Custom Functions

```
define Display
    @echo "Entering Function"
    @echo " 0 is   $0"
    @echo " 1 is   $1"
    @echo " 2 is   $2"
    @echo " @ is   $@"
    @echo "Exit Function"
endef

x := Elsayed
target1:
    $(call Display,moatasem,$(x))
    @echo
    @echo "target is $@"
```

2.2 Built-in Functions

Make provides several built-in functions:

```
# String substitution
$(subst .c,.cpp, main.c test.c lcd.c)

# Pattern substitution
$(patsubst %.c,%.o,test.c)
```

```
# Sorting
$(sort zoo bar lose)

# Word functions
$(word 3, foo bar baz)        # Gets 3rd word: "baz"
$(wordlist 2, 3, foo bar baz) # Gets words 2-3: "bar baz"
$(firstword foo bar)          # "foo"
$(lastword foo bar)           # "bar"
```

3 Conditionals and Loops

3.1 Loops

Bash-style Loops

```
LIST = one two three
loop1:
    @for i in $(LIST); do \
        echo $$i; \
    done
```

Foreach Function

```
list = foo bar baz
list_2 = $(foreach i,$(list),"\nWord is -$(i)")

loop2:
    @echo $(list_2)
```

3.2 Conditionals

```
# Equality tests (multiple syntaxes supported)
ifeq (arg1, arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"

# Inequality tests
ifneq (arg1, arg2)
ifneq 'arg1' 'arg2'

# Variable definition tests
ifdef variable-name
ifndef variable-name

# Example
bar=true
foo = bar
cond3:
ifdef $(foo)
```

```
    @echo "true"
endif
```

4 Advanced Topics

4.1 Phony Targets

Prevents conflicts between target names and file names:

```
.PHONY: clean
clean:
    @rm *.o
    @rm runprogram
```

4.2 Include Files

Split your Makefile into multiple files:

```
include makeconfig
# File 'makeconfig' contains variable definitions
```

4.3 Automatic Variables

Variable	Description
$@	The target name
$<	The first prerequisite
$	All prerequisites
$?	All prerequisites newer than target
$*	The stem of the target pattern

Table 1: Automatic Variables in Make

Example

Example using automatic variables:

```
help: info1 info2
    @echo "this is the $@"
    @echo "first prerequisite $<"
    @echo "all prerequisites $^"
```

5 Built-in Rules and Optimization

5.1 Built-in Rules

Make has implicit rules. View them with:

```
make -p      # Print database of rules
make -r      # Disable built-in rules
```

5.2 Pattern Rules

```
# Convert .c files to .o files
%.o: %.c
    $(CC) -c $< -o $@

# Usage
var2=$(file:.c=.o)
build2: $(var2)
    echo "Compile only is done"
```

5.3 Parallel Execution

Use multiple threads for faster builds:

```
make -j 4      # Use 4 parallel jobs
make -j        # Use maximum available cores
```

5.4 Debugging

Debug your Makefiles:

```
make --debug   # Show debugging information
make -n        # Dry run (show commands without executing)
```

6 Complete Example

```
# Compiler configuration
CC = gcc
CXX = g++
CFLAGS = -Wall -O2
CXXFLAGS = -Wall -O2

# Source files
C_SOURCES = test.c
CPP_SOURCES = main.cpp
OBJECTS = $(C_SOURCES:.c=.o) $(CPP_SOURCES:.cpp=.o)
EXECUTABLES = c_program cpp_program

# Default target
all: $(EXECUTABLES)

# C program
c_program: test.o
    $(CC) $(CFLAGS) $^ -o $@

# C++ program
```

```
cpp_program: main.o
    $(CXX) $(CXXFLAGS) $^ -o $@

# Pattern rule for object files
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@

# Utility targets
.PHONY: clean test display

display:
    @echo "PATH: $(PATH)"
    @echo "Compiler: $(CC)"

test:
    @python analysis.py

clean:
    @rm -f $(OBJECTS) $(EXECUTABLES)
    @echo "Cleaned on `date`" > log.txt

# Help target
help:
    @echo "Available targets:"
    @echo "  all     - Build all programs (default)"
    @echo "  clean   - Remove generated files"
    @echo "  test    - Run tests"
    @echo "  display - Show configuration"
    @echo "  help    - Show this help"
```

Listing 1: Complete Makefile Example

7 Best Practices

- **Use .PHONY**: Always declare non-file targets as phony
- **Organize**: Split complex Makefiles using include
- **Document**: Add comments explaining complex rules
- **Use variables**: Define compiler, flags, and sources as variables
- **Add help target**: Always include a help target
- **Default target**: Make 'all' the first target
- **Clean target**: Always include a clean target
- **Use pattern rules**: For efficient compilation

- **Parallel builds**: Enable with -j flag
- **Error handling**: Use '-' prefix to ignore errors when appropriate

8 Quick Reference

Command	Purpose
make	Build default target
make target	Build specific target
make -f filename	Use alternate makefile
make -j N	Parallel build with N jobs
make -n	Dry run (show commands)
make –debug	Debug mode
make -r	Disable built-in rules
make -p	Print rule database

Table 2: Make Command Quick Reference