GNU Make

Complete Guide to Makefiles and Build Automation

Build Automation Tool

From Basic Compilation to Complex Build Systems

Type: Build automation tool
Configuration: Makefile
Interface: Command line
Invented: 1976 by Stuart Feldman
Variant: GNU Make (most common)
Standard: POSIX make

Version 4.3 | December 24, 2025

make | Makefile | Rules | Variables | Automation

Contents

1 What is Make?

1.1 Definition

Make - The Build Automation Tool

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.

1.2 Key Characteristics

- **Dependency-based**: Builds targets based on their dependencies
- **Incremental**: Only rebuilds what's necessary
- **Declarative**: You specify *what* to build, not *how*
- **Platform-independent**: Works on Unix, Linux, macOS, Windows (with ports)
- **Language-agnostic**: Can build C, C++, Java, Python, LaTeX, etc.

1.3 History and Variants

Variant	Description
GNU Make	Most popular, extended features, Linux standard
BSD Make	Used on BSD systems (FreeBSD, OpenBSD)
Microsoft NMake	Microsoft's version for Windows
Sun Make	Original Unix make from Sun Microsystems
CMake	Meta-build system that generates Makefiles

Table 1: Make Variants

2 Why Do We Need Make?

2.1 Problems Without Make

```
# Compiling a simple C program manually
gcc -c main.c -o main.o
gcc -c utils.c -o utils.o
gcc -c math.c -o math.o
gcc main.o utils.o math.o -o program

# After changing utils.c:
gcc -c utils.c -o utils.o       # Have to remember which file changed
```

```
gcc main.o utils.o math.o -o program  # Have to remember to relink

# After adding new file network.c:
gcc -c main.c -o main.o
gcc -c utils.c -o utils.o
gcc -c math.c -o math.o
gcc -c network.c -o network.o    # Forgot this step initially!
gcc main.o utils.o math.o network.o -o program
```

Listing 1: Manual Compilation - The Hard Way

2.2 Benefits of Using Make

- **Automation**: No manual compilation commands
- **Incremental builds**: Faster compilation
- **Parallel builds**: Build multiple files simultaneously
- **Dependency tracking**: Knows what needs rebuilding
- **Consistency**: Same build process every time
- **Portability**: Works across different systems
- **Complex builds**: Handle complex dependencies
- **Clean builds**: Easy cleanup of generated files
- **Installation**: Standard install/uninstall targets
- **Documentation**: Makefile documents build process

2.3 Real-World Example: Without vs With Make

```
# WITHOUT MAKE (shell script build.sh)
#!/bin/bash
echo "Compiling source files..."
gcc -c main.c -o main.o -I./include -Wall
gcc -c utils.c -o utils.o -I./include -Wall
gcc -c network.c -o network.o -I./include -Wall
gcc -c gui.c -o gui.o -I./include -Wall
gcc main.o utils.o network.o gui.o -o myapp -lm -lpthread

echo "Cleaning..."
rm -f *.o myapp

# WITH MAKE (Makefile)
CC = gcc
CFLAGS = -I./include -Wall
LDFLAGS = -lm -lpthread
OBJS = main.o utils.o network.o gui.o

myapp: $(OBJS)
    $(CC) -o $@ $^ $(LDFLAGS)

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

```

```
clean:
    rm -f $(OBJS) myapp
```

Listing 2: Manual Build Script vs Makefile

3 Is It Mandatory to be Called Makefile?

3.1 Default Names

Default Makefile Names

Make looks for build configuration files in this order:

1. `GNUmakefile` (GNU Make specific)
2. `makefile` (lowercase)
3. `Makefile` (uppercase M, most common)

3.2 Specifying Different Names

```
# Using a non-standard Makefile name
make -f build.mk
make --file=build.mk
make --makefile=build.mk

# Using multiple Makefiles
make -f Makefile.linux
make -f Makefile.windows
make -f Makefile.debug
make -f Makefile.release

# In a Makefile, include others
include config.mk
include rules.mk
include platforms/linux.mk
```

Listing 3: Using Different Makefile Names

3.3 Why ”Makefile” is Standard

- **Convention**: Established practice since 1970s
- **Visibility**: Easy to spot in directory listings
- **Case-sensitive**: Works on all filesystems
- **Tool integration**: IDEs and editors recognize it
- **Documentation**: Clear it’s a build configuration

4 Basic Syntax

4.1 Makefile Structure

```
# Comments start with '#'
# Variables
CC = gcc
CFLAGS = -Wall -O2

# Rules
target: dependencies
    command1
    command2
    command3

# Another rule
another_target: another_dependencies
    command

# Phony targets (not files)
.PHONY: clean install
```

Listing 4: Basic Makefile Structure

5 Setting Rules

5.1 Basic Rule Syntax

```
# Simple rule
target: prerequisite1 prerequisite2 prerequisite3
    recipe line 1
    recipe line 2
    recipe line 3

# Example: Compile C program
program: main.o utils.o
    gcc main.o utils.o -o program

main.o: main.c
    gcc -c main.c -o main.o

utils.o: utils.c
    gcc -c utils.c -o utils.o

# Pattern rule
%.o: %.c
    gcc -c $< -o $@

# Multiple targets
all: program1 program2 program3

program1: file1.o
    gcc file1.o -o program1

```