GNU Make

Build automation per progetti C/C++ su sistemi *NIX

Roberto Marino, PhD

Università di Messina
CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

April 23, 2026



UniMe
1548

Cos'e' GNU Make?

- **Make** e' uno strumento di build automation
- Legge un file chiamato `Makefile` e decide *quali comandi eseguire*
- Creato da **Stuart Feldman** nel **1976** ai Bell Labs
- Oggi lo standard de facto su sistemi UNIX/Linux e' **GNU Make**

Problema che risolve

Compilare a mano un progetto con decine di file e' lento, error-prone e non riproducibile.

Senza Make

```
gcc -c main.c -o main.o
gcc -c utils.c -o utils.o
gcc -c parser.c -o parser.o
gcc main.o utils.o \
    parser.o -o prog
```

Con Make

```
make
```

Perche' usare Make?

Vantaggi principali

- **Build incrementale**
  ricompila *solo* i file modificati
- **Automazione**
  un solo comando per tutto
- **Riproducibilita'**
  stesso risultato su ogni macchina
- **Documentazione implicita**
  il Makefile descrive come si costruisce il progetto

Non solo compilazione

- Eseguire test automatici
- Generare documentazione (Doxygen, LaTeX...)
- Deploy e installazione
- Pulizia dei file generati
- Qualsiasi pipeline ripetibile

La Struttura Base di una Regola

Sintassi fondamentale

```
target: dipendenza1 dipendenza2
    ricetta_comando1
    ricetta_comando2
```

[T] Target

Il file (o azione) da produrre.
Es: `main.o`, `programma`

[D] Dipendenze

I file necessari. Se cambiano, il target viene ricostruito.

[R] Ricetta

I comandi shell da eseguire.
DEVONO iniziare con TAB!

Esempio concreto

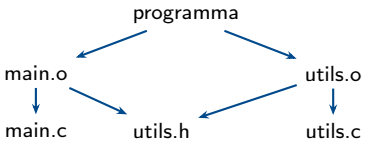
```
main.o: main.c utils.h
    gcc -c main.c -o main.o

programma: main.o utils.o
    gcc main.o utils.o -o programma
```

Come Funziona Make Internamente

Algoritmo di Make

1. Legge il Makefile e costruisce un **DAG** (grafo aciclico diretto) delle dipendenze
2. Per ogni target, confronta il **timestamp** del file con quello delle sue dipendenze
3. Se una dipendenza e' *piu' recente* del target (o il target non esiste) $\Rightarrow$ esegue la ricetta
4. Risale l'albero ricorsivamente finche' tutto e' aggiornato



Build incrementale

Se modifico solo `utils.c`, Make ricompila *solo* `utils.o` e `programma`. Non tocca `main.o`.

Variabili

Definizione e utilizzo

```
CC      = gcc
CFLAGS  = -Wall -Wextra -g
TARGET  = programma
OBJS    = main.o utils.o

$(TARGET): $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)

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

Tipi di assegnazione

=	Lazy: espanso ogni volta che usata
:=	Immediata: espanso subito
?=	Solo se non gia' definita
+=	Appende al valore

Override dalla CLI

```
make CC=clang CFLAGS="-O2"
```

Target Phony

Cosa sono i target phony?

Un target **phony** non corrisponde a nessun file reale: e' solo il nome di un'azione.

Dichiarazione

```
.PHONY: all clean install test

all: $(TARGET)

clean:
    rm -f $(OBJS) $(TARGET)

install: $(TARGET)
    cp $(TARGET) /usr/local/bin/
```

Regola d'oro

Il **primo target** del Makefile e' quello eseguito di default con `make` senza argomenti.

Per convenzione si chiama `all`.

Target comuni

`all`	Build principale
`clean`	Rimuove gli artifact
`install`	Copia in destinazione
`test`	Esegue i test
`debug`	Build con -g
`help`	Stampa l'aiuto

Pattern Rules e Variabili Automatiche

Pattern Rules

Il carattere % e' una wildcard che matcha qualsiasi stringa.

```
# Senza pattern rule (ridondante):
main.o: main.c
    gcc -c main.c -o main.o
utils.o: utils.c
    gcc -c utils.c -o utils.o

# Con pattern rule (elegante):
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

Variabili Automatiche

$@	Nome del target in costruzione
$<	Prima dipendenza
$^	Lista di tutte le dipendenze
$*	Stem (parte della regola che matchato con %)
$?	Dipendenze piu' recenti

Esempio

```
prog: a.o b.o c.o
    $(CC) $^ -o $@
# gcc a.o b.o c.o -o prog
```

Funzioni Built-in

Le funzioni piu' utili

```
# Trova tutti i .c in src/
SRCS := $(wildcard src/*.c)

# Trasforma path
OBJS := $(patsubst src/%.c, \
            build/%.o, $(SRCS))

# Esegue un comando shell
GIT  := $(shell git rev-parse HEAD)

# Rimuove un elemento
LIBS := $(filter-out main.o, $(OBJS))

# Ordina e deduplica
SORTED := $(sort $(OBJS))
```

Funzioni su path

```
# Estrae il nome del file
$(notdir src/main.c)
# -> main.c

# Estrae la directory
$(dir src/main.c)
# -> src/

# Rimuove l'estensione
$(basename main.c)
# -> main
```

Direttive Condizionali

ifeq / ifdef

```
DEBUG ?= 0
ifeq ($(DEBUG), 1)
    CFLAGS += -g -DDEBUG
    CFLAGS += -fsanitize=address
else
    CFLAGS += -O2 -DNDEBUG
endif

ifdef PREFIX
    INSTALL_DIR = $(PREFIX)/bin
else
    INSTALL_DIR = /usr/local/bin
endif
```

include

```
# Includere altri Makefile
include config.mk

# Ignorare errore se assente
-include .deps/*.d
```

Uso dalla riga di comando

```
# Build normale
make

# Build di debug
make DEBUG=1

# Installa in path custom
make PREFIX=/opt install
```

Il Problema degli Header File

Problema classico

Se `main.c` include `utils.h` e quest'ultimo cambia, occorre ricompilare `main.c`. Gestirlo a mano e' error-prone e non scala.

Soluzione: flag GCC `-MMD -MP`

```
CC      := gcc
CFLAGS  := -Wall -MMD -MP
SRCS    := $(wildcard src/*.c)
OBJS    := $(patsubst src/%.c, \
               build/%.o, $(SRCS))
DEPS    := $(OBJS:.o=.d)

-include $(DEPS)

build/%.o: src/%.c | build
    $(CC) $(CFLAGS) -c $< -o $@

build:
    mkdir -p $@
```

File .d generato da GCC

```
# build/main.d (auto-generato)
build/main.o: src/main.c \
    include/utils.h \
    include/config.h
```

Cosa fanno i flag

- `-MMD`: genera il file `.d` con le dipendenze dagli header
- `-MP`: aggiunge target vuoti per evitare errori se un header viene eliminato
- `-include`: il trattino ignora l'assenza dei file al primo build

Struttura di un Progetto Reale

Layout directory

```
mio_progetto/
+-- Makefile
+-- src/
|   +-- main.c
|   +-- parser.c
|   `-- utils.c
+-- include/
|   +-- parser.h
|   `-- utils.h
`-- build/
    +-- main.o
    +-- parser.o
    `-- utils.o
```

Makefile completo

```
CC       := gcc
CFLAGS   := -Wall -Wextra -std=c11
CPPFLAGS := -Iinclude
TARGET   := programma
BUILD    := build
SRCS     := $(wildcard src/*.c)
OBJS     := $(patsubst src/%.c, \
                $(BUILD)/%.o, $(SRCS))
DEPS     := $(OBJS:.o=.d)

.PHONY: all clean install debug

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $^ -o $@

$(BUILD)/%.o: src/%.c | $(BUILD)
    $(CC) $(CFLAGS) $(CPPFLAGS) \
          -MMD -MP -c $< -o $@

$(BUILD):
    mkdir -p $@

clean:
    rm -rf $(BUILD) $(TARGET)

debug: CFLAGS += -g -DDEBUG
debug: all

-include $(DEPS)
```