SAPIENZA
UNIVERSITÀ DI ROMA

Ingegneria Informatica e
Automatica

Tecniche di Programmazione

A.A. 2017/2018

Introduzione ai makefile e all'utility GNU make. Cenni su Cmake.

Introduzione (1/2)

- Il codice sorgente dei programmi è solitamente diviso in più file. → Per creare l'eseguibile è necessario **compilare** ogni singolo sorgente e **linkare** tutti i file oggetto ottenuti.
- In questi casi, non è una buona idea compilare usando la semplice linea di comando:

gcc <opzioni> -o eseguibile source1.c ... sourceN.c

In caso di modifica di un solo sorgente (es. **sourceX.c**), non è necessario ricompilare anche tutti gli altri sorgenti. Si pensi al caso di un progetto composto da N>100 file sorgenti!!

Introduzione (2/2)

- Soluzione: individuare i file che sono stati modificati dall'ultima compilazione, ricompilare soltanto questi ultimi e rieseguire il processo di linking tra tutti i file oggetto.
- A tale scopo, è necessario definire una serie di **regole** che definiscono le dipendenze che intercorrono tra i vari sorgenti/header e i comandi necessari per la compilazione e la creazione dell'eseguibile. Il file che contiene tali regole è chiamato **makefile.**
- I makefile verranno passati come input ad una utility (in Linux **GNU make**) che, in maniera automatica:
  - determina quali componenti sono state modificate;
  - esegue di conseguenza i comandi (es. compilazione e linking)

I makefile

- Il makefile di un progetto software è di solito memorizzato in un file di nome **Makefile**.
- Le righe che iniziano con **#** sono commenti.
- La sintassi dei makefile è molto rigida. Ogni direttiva deve stare in una singola riga, per spezzare una direttiva su più linee è necessario utilizzare il carattere **** ad esempio:

  **CC -o nome_eseguibile main.o points.o lines.o **
  squares.o circles.o

  senza aggiungere ulteriori spazi.
- I costrutti fondamentale di un makefile sono le **rules** (regole)

Le regole

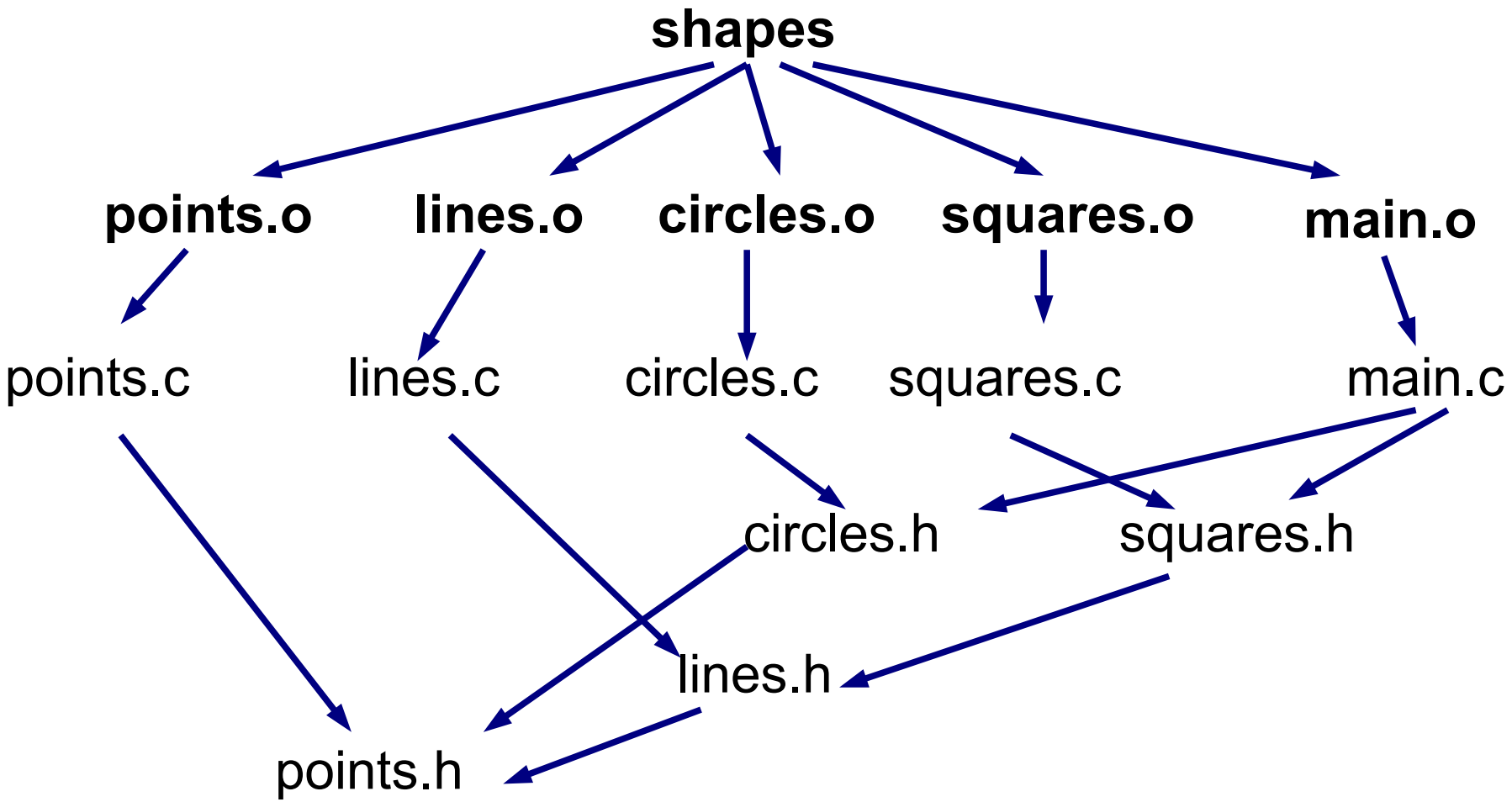
- Una regola ci dice **quando** e **come** ottenere un certo file obiettivo (target), ad esempio un file oggetto, un eseguibile (o un'azione).
- Sintassi:
  TARGET : DIPENDENZE
  <tab>COMANDO
  - TARGET: di solito il nome dell'eseguibile o dell'oggetto che si vuole ottenere dalla compilazione, oppure l'azione che si vuole compiere.
  - DIPENDENZE: i file o altri TARGET da cui dipende il TARGET della regola. Le dipendenze ci dicono **quando** ottenere TARGET.
  - COMANDO: il comando da eseguire, ovvero ci dice **come** ottenere TARGET.

Un esempio (1/3)

- Supponiamo di avere i 4 file sorgenti points.c, lines.c, squares.c, circles.c e rispettivi header (*.h) ed il sorgente main.c che contiene la funzione main(). Vogliamo preparare un makefile per generare l'eseguibile **shapes**.
- Analizzando il codice, si può notare che points.c include points.h (ove sono dichiarate le funzioni definite in points.c), così come lines.c include lines.h e così via. Inoltre lines.h, e circles.h includono points.h, mentre squares.h include lines.h (concettualmente: una linea può essere definta da due punti, quindi nelle funzioni di lines.h dovranno essere utilizzati costrutti di points.h e così via). main.c, infine, include squares.h e circles.h.

Un esempio (2/3)

- Schematizziamo le dipendenze:



Un esempio (3/3)

- Il makefile risultante:

```
shapes : main.o points.o lines.o circles.o squares.o
	gcc -o shapes main.o points.o lines.o circles.o squares.o
points.o : points.c points.h
	gcc -c points.c
lines.o : lines.c lines.h points.h
	gcc -c lines.c
circles.o : circles.c circles.h points.h
	gcc -c circles.c
squares.o : squares.c squares.h lines.h points.h
	gcc -c squares.c
main.o : main.c squares.h circles.h lines.h points.h
	gcc -c main.c
```

Le azioni

- Dichiarando un target come .PHONY comunichiamo a **make** che quel particolare nome di target non rappresenta un file.
- Esempio: aggiungendo le seguenti righe al precedente makefile (alla fine):

```
.PHONY: clean
clean :
	rm -f shapes main.o points.o lines.o circles.o squares.o
```

Lanciando **make clean** verranno eliminati tutti i file oggetto e l'eseguibile eventualmente creati in precedenza.

GNU make

- Il comando **make** legge di default il file Makefile posto nella directory corrente.
- La sintassi del comando è:
  make [<opzioni>] [target]
- **make** processa di default la prima regola che trova nel makefile, nel caso dell'esempio **shapes** e, in maniera ricorsiva, tutte le regole che hanno come target una delle dipendenze della regola appena processata.
- Per cambiare la directory ove cercare il file Makefile, usare l'opzione **-C <directory>.**
- **make** si ferma ogni volta che l'esecuzione di un comando di una regola si blocca a causa di un errore (esempio, di compilazione). Utilizzare l'opzione **-k** per proseguire con le regola successive.

Le variabili (1/2)

- Utili per evitare di ripetere in più parti del makefile argomenti (ad esempio percorsi degli header e delle librerie) e liste di file.
- Definire una variabile:
  NOME_VARIBILE=<lista argomenti>
- Richiamare una variabile:
  $(NOME_VARIABILE)
- Nell'esempio precedente:

```
OBJECTS=main.o points.o lines.o circles.o squares.o
shapes : $(OBJECTS)
	gcc -o shapes $(OBJECTS)
# ....
clean :
	rm -f shapes $(OBJECTS)
```

Le variabili (2/2)

- Alcune variabili automatiche (da usarsi all'interno delle regole):
  $@ : File target di una regola
  $^ : Elenco di tutte le dipendenze
  $< : la prima dipendenza di una regola

Variabili automatiche: un esempio

```
CC = gcc -g -Wall
OBJECTS=test_list.o part_list.o
.PHONY: clean
test_list : $(OBJECTS)
	$(CC) -o $@
#Equivalente a:
#gcc -g -Wall -o test_list test_list.o part_list.o

part_list.o : part_list.c part_list.h
	$(CC) -c $<
#Equivalente a:
#gcc -g -Wall -c part_list.c

test_list.o : test_list.c part_list.h
	$(CC) -c $<
#Equivalente a:
#gcc -g -Wall -c test_list.c

clean :
	rm -f test_list $(OBJECTS)
```

Dipendenze su più righe

- Per un singolo target, è possibile definire le sue dipendenze utilizzando più regole, ognuna delle quali condivide tale target.
- **Una sola di queste deve però definire il comando per il target in questione**:
- Esempio:

```
TARGET_X : DIPENDENZE_1
<tab>COMANDO

TARGET_X : DIPENDENZE_2
TARGET_X : DIPENDENZE_3
...
```

Creazione automatica delle dipendenze (1/3)

- Il compilatore GNU C mette a disposizione il comando **gccmakedep** il quale, per ogni sorgente passato in input, modifica il makefile aggiungendo alla fine dello stesso la lista delle dipendenze automaticamente dedotte per il sorgente in questione.
- Sintassi:
  - gccmakedep -- <options> -- sourcefiles (ove options sono le opzioni di compilazione, ad esempio le directory ove si trovano gli header, es. -I/usr/include).

Creazione automatica delle dipendenze (2/3)

- Normalmente gccmakedep viene utilizzato aggiungendo un target di tipo .PHONY, ad esempio di nome depend:
  - .PHONY depend
    SRCS = file1.c file2.c …
    CFLAGS = -I/usr/include -I./include
    depend:
        gccmakedep -- $(CFLAGS) -- $(SRCS)
- Invocando make depend le dipendenze verranno aggiornate.
- Gccmakedep aggiunge la lista delle dipendenze alla fine del makefile, di seguito al commento automaticamente generato: # DO NOT DELETE.

Creazione automatica delle dipendenze (3/3)

- Nell'esempio precedente:

```
CC = gcc -g -Wall
OBJECTS=test_list.o part_list.o
.PHONY: clean depend
# …..
depend:
	gccmakedep -- -- part_list.c test_list.c
```

- Invocando make depend:

```
# …..
depend:
	gccmakedep -- -- part_list.c test_list.c# DO NOT DELETE
part_list.o: part_list.c /usr/include/stdio.h /usr/include/features.h \
 /usr/include/bits/predefs.h /usr/include/sys/cdefs.h \
 /usr/include/bits/wordsize.h /usr/include/gnu/stubs.h \
 /usr/include/gnu/stubs-64.h \
 /usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stddef.h \ ….
```

Cenni su CMake: Cross-Platform Make

Motivazioni (1/2)

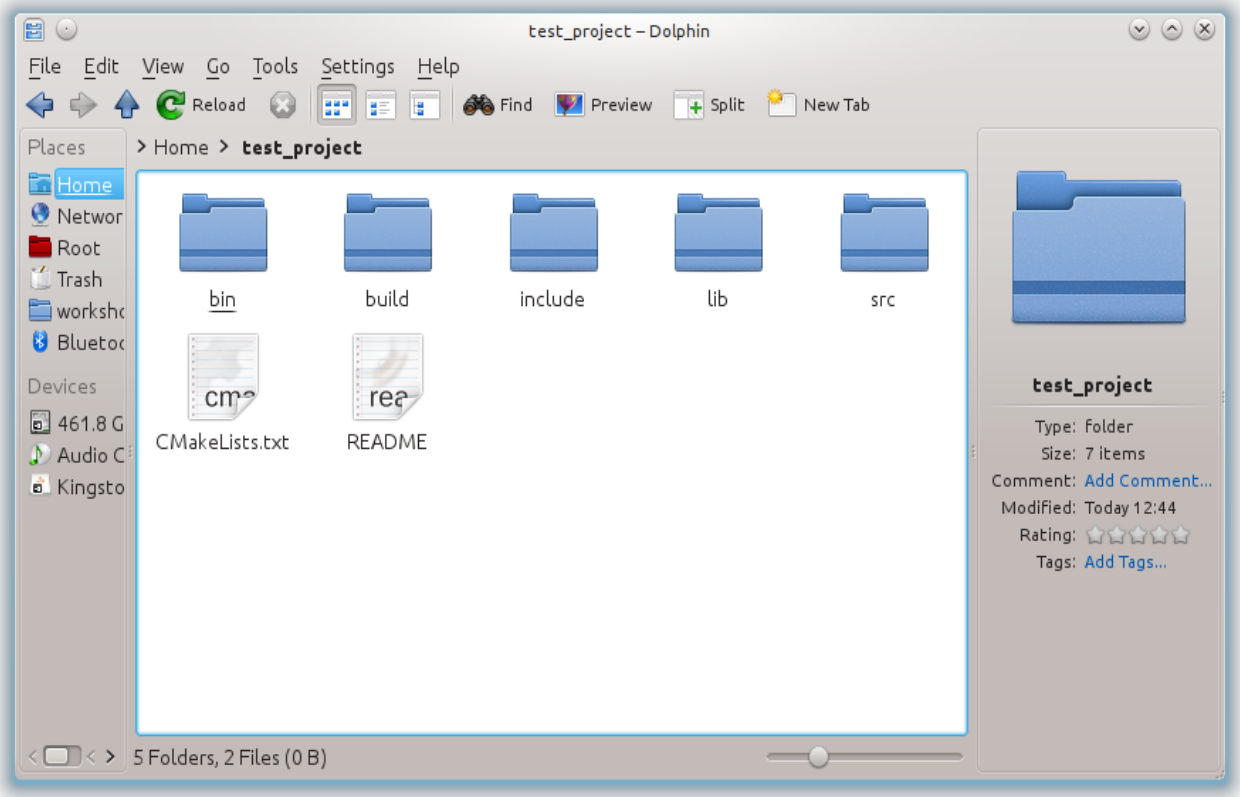
- GNU Make permette di gestire la compilazione di programmi complessi, composti da decine/centinaia di sorgenti e dipendenti da decine di librerie, ma:
  - E' spesso necessario gestire manualmente le dipendenze;
  - E' necessario conoscere la posizione precisa delle librerie linkate e dei rispettivi header;
  - La sintassi dei makefile è piuttosto complessa e forse eccessivamente rigida;
  - GNU Make funziona sui sistemi Unix: altri sistemi operativi (es. Windows) hanno un diverso tool per gestire la compilazione di programmi, con funzionalità molto simili a GNU Make ma con makefile dalla sintassi completamente diversa.

Motivazioni (2/2)

- CMake (Cross-Platform Make) è un tool che automatizza il processo di creazione dei makefile, semplificando la compilazione di progetti complessi:
  - **CMake non sostituisce GNU Make**: esso infatti, seguendo un file di configurazione formato testo editato dallo sviluppatore (CMakeLists.txt), crea automaticamente un file Makefile da utilizzare in seguito con il comando make per compilare il progetto.
  - CMake è cross-platform: utilizzando lo stesso file di configurazione (CMakeLists.txt) su diversi sistemi operativi, CMake genera diversi makefile dipendentemente dal sistema operativo in cui è stato lanciato.
  - Cmake permette di semplificare ed automatizzare diverse operazioni attraverso una sintassi semplice ed intuitiva.

Un esempio (1/3)

- Vogliamo creare un semplice progetto software ben strutturato:



Un esempio (2/3)

- Contenuto di una possibile cartella di progetto:
  - Cartella **src/**: contiene i sorgenti (.c) del progetto, es, source1.c, source2.c e source3.c.
  - Cartella **include/**:contiene gli header (.h) del nostro progetto, es. header1.h, header2.h e header3.h.
  - Cartella **lib/**: qui verranno eventualmente messe le librerie create.
  - Cartella **bin/**: qui verranno messi gli eseguibili creati
  - Cartella **build/**: qui verranno messi i file generati da Cmake, tra cui il Makefile automaticamente generato.
  - File **CMakeLists.txt**: file di configurazione per Cmake
  - File **README**: breve presentazione del software e istruzioni di compilazione.

Un esempio (3/3)

- Supponiamo che tale progetto faccia uso di funzioni delle librerie GTK2, il file CMakeLists.txt potrebbe essere:

```
project( test_project C )

cmake_minimum_required( VERSION 3.0 )

find_package( GTK2 2.9 REQUIRED )

set( SRCS src/source1.c src/source2.c src/source3.c )

include_directories( include ${GTK2_INCLUDE_DIRS} )

add_executable( my_app ${SRCS} )
target_link_libraries( my_app ${GTK2_LIBRARIES} )

set_target_properties( my_app PROPERTIES RUNTIME_OUTPUT_DIRECTORY
                       ${PROJECT_SOURCE_DIR}/bin )
```

CMakeLists.txt passo-passo (1/2)

```
# Nome del progetto e linguaggio supportato
project( test_project C )

# Definisce la versione minima supportata da CMake per questo progetto
cmake_minimum_required( VERSION 3.0)

# Il comado find_package() comunicare a CMake di cercare automaticamente
# una libreria, qui richiedo le librerie GTK2 nella versione 2.9 o successiva
find_package( GTK2 2.9 REQUIRED  )

# Il comado set() serve per definire nuove variabili.
# Qui definisco la variabile SRCS che conterrà la lista dei sorgenti da compilare
set( SRCS src/source1.c src/source2.c src/source3.c )

# Il comado include_directories() notifica a CMake quali directory conterranno gli header
# file (oltre ai path di default, es /usr/include). La precedente chiamata di find_package
# ha automaticamente settato la variabile GTK2_INCLUDE_DIRS che contiene la lista
# delle directory in cui si trovano gli header delle librerie gtk.
# Devo aggiungere anche la directory include locale.
include_directories( include ${GTK2_INCLUDE_DIRS} )
```

CMakeLists.txt passo-passo (2/2)

```
# Il comando add_executable() comunica a CMake di generare un eseguibile a
# partire da una lista di sorgenti. Qui richiedo di generare un eseguibile di nome my_app
# a partire dalla sequenza di file sorgenti rappresentati dalla variabile SRCS
# definta prima
add_executable( my_app ${SRCS} )

# Il comando target_link_libraries() comunica a CMake quali librerie linkare per generare
# un certo eseguibile. La precedente chiamata di find_package ha automaticamente
# settato la variabile GTK2_LIBRARIES che contiene la lista delle librerie GTK2.
# Per ottenere l'applicazione my_app ad esempio devo linkare anche le librerie GTK2.
target_link_libraries( my_app ${GTK2_LIBRARIES} )

# Il comando set_target_properties() serve per modifcare varie impostazioni di default.
# Qui sto semplicemente richiedendo di posizionare l'eseguibile risultato della compilazione
# all'interno della directory bin locale.
set_target_properties( my_app PROPERTIES RUNTIME_OUTPUT_DIRECTORY $
                       {PROJECT_SOURCE_DIR}/bin )
```

Per creare una libreria

```
# Il comando add_library() comunica a CMake di generare una librerie a partire da una
# lista di sorgenti. Qui richiedo di generare una libreria statica di nome my_lib a partire
# dalla sequenza di file sorgenti rappresentati dalla variabile SRCS defintа prima
add_library( my_lib STATIC ${SRCS} )

# Il comando set_target_properties() serve per modifcare varie impostazioni di default.
# Qui sto semplicemente richiedendo di posizionare la librera risultato della compilazione
# all'interno della directory lib locale.
set_target_properties( my_lib PROPERTIES ARCHIVE_OUTPUT_DIRECTORY $
                                {PROJECT_SOURCE_DIR}/lib )
```

Utilizzo base di CMake

- Installazione di CMake:
  - Da terminale lanciare: **sudo apt-get install cmake**
- Lanciare CMake per generare il makefile:
  - Spostarsi sulla directory **build** e lanciare **cmake ..**
- Se CMake non ha generato errori, all'inerno di **build**, oltre ad alcuni file utilizzati da CMake stesso, vi sarà un Makefile compatibile con GNU Make. Lanciare a questo punto **make** per la compilazione vera e propria. Se non vi sono errore di compilazione, potremo trovare l'eseguibile richiesto all'iterno della directory **bin**.