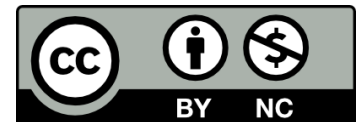


GNU Make, Autotools, CMake 簡介

Wen Liao



Disclaimer

投影片資料為作者整理資料及個人意見，沒有經過嚴謹確認，請讀者自行斟酌

目標

- 簡介GNU Make, Autotools, 和CMake
 - Autotools和CMake部份極度簡介

測試環境

```
$ lsb_release -a  
No LSB modules are available.  
Distributor ID: Ubuntu  
Description:    Ubuntu 12.04.4 LTS  
Release:        12.04  
Codename:       precise
```

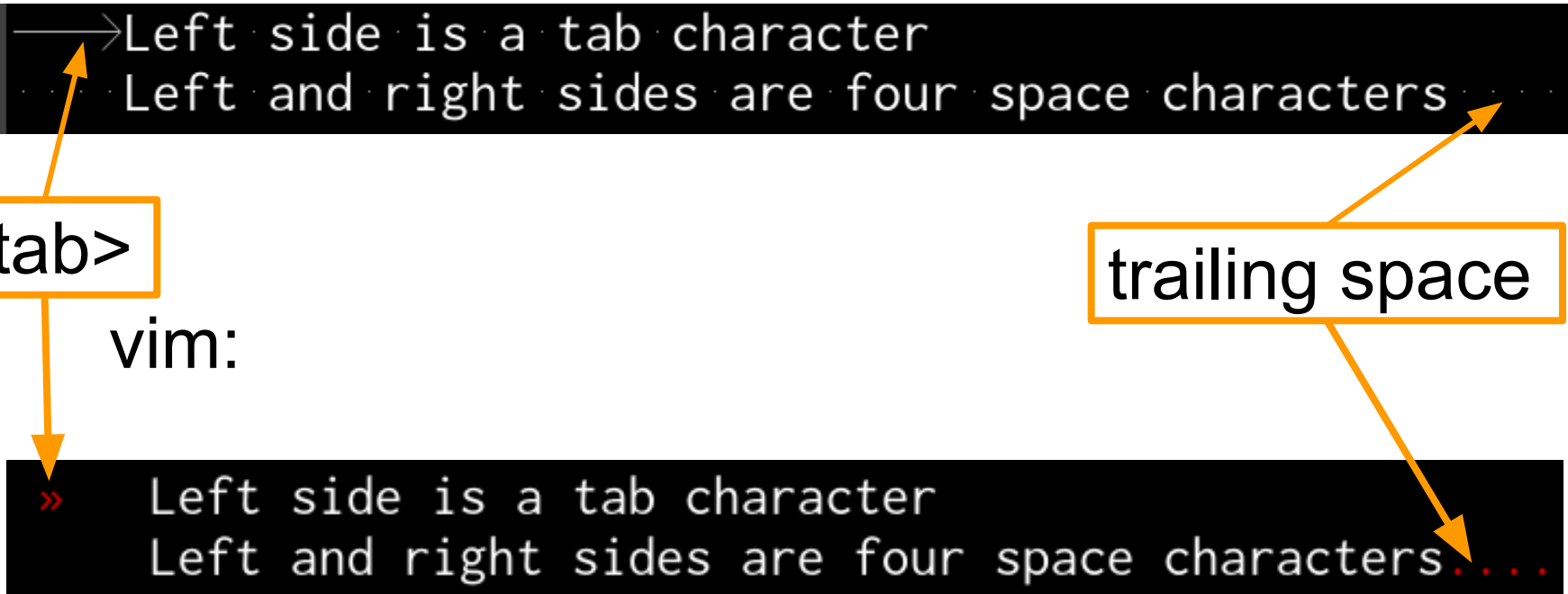
Outline

- **GNU Make**
- Autotools
- CMake
- 參考資料
- Q & A

事先準備

- 為了您的生活幸福美滿，請確認編輯器支援顯示空白字元和<tab>字元!

geany:



→ Left side is a tab character
Left and right sides are four space characters

The diagram shows two terminal windows. The top window, labeled 'geany:', displays two lines of text. An orange arrow points from a box containing '<tab>' to the start of the first line. Another orange arrow points from a box containing 'trailing space' to the end of the second line. The bottom window, labeled 'vim:', shows the same text but with a red '>>' at the start of the first line, indicating the tab character is visible. The 'trailing space' box also has an arrow pointing to the end of the second line in this window.

<tab>

trailing space

vim:

>> Left side is a tab character
Left and right sides are four space characters....

關於GNU Make

- man make
 - GNU make utility to maintain groups of programs
 - ???

直接看例子

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello world\n");

    return 1;
}
```

怎麼編譯？

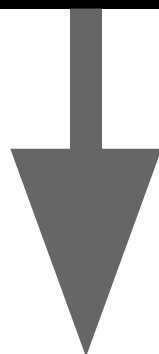
簡單, gcc hello.c

加碼，這個怎麼編譯？

```
$ tree
.
├── include
│   ├── liba.h
│   └── libb.h
├── libs
│   ├── liba.c
│   └── libb.c
└── src
    └── test.c
```

GG! 還要加include Path

```
$ gcc src/test.c libs/liba.c libs/libb.c  
src/test.c:1:18: fatal error: libb.h: No such file or directory  
compilation terminated.  
libs/liba.c:1:18: fatal error: libb.h: No such file or directory  
compilation terminated.  
libs/libb.c:1:18: fatal error: liba.h: No such file or directory  
compilation terminated.
```



```
$ gcc src/test.c libs/liba.c libs/libb.c -I./include
```

那麼上百個檔案怎麼辦？打到死？

```
$ cloc .  
defined(%hash) is deprecated at /usr/bin/cloc line 1277.  
    (Maybe you should just omit the defined()?)  
1613 text files.  
1287 unique files.  
844 files ignored.  
  
http://cloc.sourceforge.net v 1.53  T=2.0 s (375.5 files/s, 187967.0 lines/s)  
-----  
Language                files            blank            comment              code  
-----  
C                        256             29382             39318             122500
```

專案檔案量大編譯的問題

- 我能不能打一行指令就幫我自動編譯？
- 我能不能只編譯更動過的檔案？
 - 包含改了*.h檔案對應*.c都可以自動重編
- 可不可以有沒有靈活的編譯組態？
 - 設定debug mode還是release mode
 - 設定編譯器選項
 - Compile time指定巨集
 - -DMY_VAR=1
 - ...

Yes You Can!

關於GNU Make

- man make
 - The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them.
 - 白話: 協助編譯的時候決定
 - 那部份要重編
 - 指定重新編譯的動作

Hello Makefile



以下情況會執行**recipe**

- 新增:**target**檔案不存在
 - 通常**target**是一個**檔案**, 但是這不是必要條件
- 更新:**prerequisites**檔案**更動時間**比**target**檔案還要新

誰說Makefile一定要編譯檔案？

target

prerequisites

```
test_file: dep_file
```

recipe

```
→ @echo Test
```

```
$ ls
Makefile
$ make
make: *** No rule to make target 'dep_file', needed by 'test_file'.  Stop.
$ touch dep_file
$ make
Test
$ touch test_file
$ make
make: 'test_file' is up to date.
$ touch dep_file
$ make
Test
```

一開始只有Makefile, 下make出現錯誤說無法產生dep_file

產生dep_file後執行@echo Test

產生test_file就不執行@echo Test

更新dep_file 更動時間後執行@echo Test

樹狀target

```
root_taget: sub_target1 sub_target2
——>@echo $@

sub_target1:
——>@echo $@

sub_target2:
——>@echo $@
```

- 第一個target稱為default target, 也是make的進入點。
- @表示不要把指令印出, 可以練習把他拿掉看看。
- \$@展開後是target的名稱

請比對上頁，觀察順序！

```
$ make  
sub_target1  
sub_target2  
root_target
```

變數

- 設定

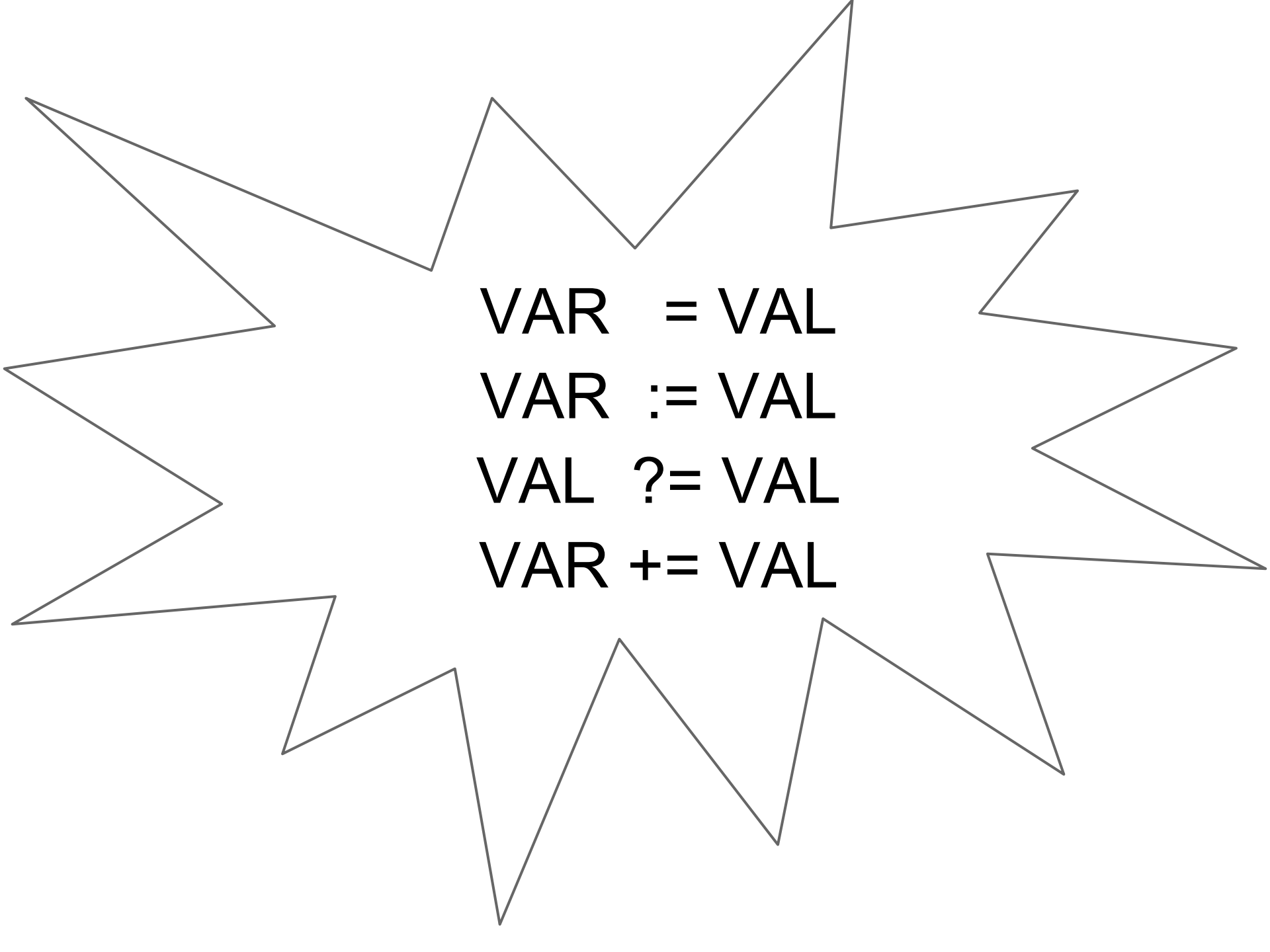
- VAR = VAL
- VAR := VAL
- VAL ?= VAL
- VAR += VAL
 - 其他我不懂的

- 設定時機

- 檔案, 通常就是Makefile內
- make 命令的參數
- 環境變數

- 取值

- \$(VAR)



VAR = VAL

VAR := VAL

VAL ?= VAL

VAR += VAL

三小？

變數設定：連動型 =

```
VAR1=first
```

VAR1 第一次設定

```
VAR2=${VAR1}
```

VAR2 的值和VAR1相同

```
$(warning ${VAR2})
```

印出VAR2的內容

```
VAR1=second
```

VAR1 第二次設定

```
$(warning ${VAR2})
```

再印出VAR2的內容

```
fake:
```

```
→@echo "Do something"
```

兩次VAR2的內容，會隨VAR1改變

```
$ make  
Makefile:3: first  
Makefile:6: second  
Do something
```


變數設定：立刻生效型 :=

```
VAR1=first  
VAR2:=$(VAR1)  
$(warning $(VAR2))
```

只改這行，把 = 改成 :=

```
VAR1=second  
$(warning $(VAR2))
```

```
fake:  
→@echo "Do something"
```

```
$ make  
Makefile:3: first  
Makefile:6: first  
Do something
```

變數設定：預設型 ?=

```
VAR1?=default  
$(warning ${VAR1})  
  
fake:  
→@echo "Do something"
```

```
$ make  
Makefile:2: default  
Do something  
$ make VAR1=Test  
Makefile:2: Test  
Do something
```

變數設定：加碼型 +=

```
VAR1=first  
$(warning ${VAR1})
```

```
VAR1+=second  
$(warning ${VAR1})
```

```
fake:  
——>@echo "Do something"
```

```
$ make  
Makefile:2: first  
Makefile:5: first second  
Do something
```

小結

設定	理解方式
VAR = VAL	連動形
VAR := VAL	立即生效形
VAL ?= VAL	預設形
VAR += VAL	加碼形
其他 <u>我不懂的</u>	不懂的不知道怎麼解釋

內建變數 (節錄)

名稱	意義
<code>\$@</code>	target名稱
<code>\$^</code>	所有的prerequisites名稱
<code>\$<</code>	第一個prerequisite名稱 用途之一： target: dep1.c inc.h test.h <tab> gcc -o <code>\$@</code> <code>\$<</code>
<code>\$?</code>	比target還新的prerequisites名稱

範例

```
TARGET=hello
SRCS=hello.c
CFLAGS=-g -Wall -Werror
```

設定要產生的執行檔

設定要編譯的程式檔

編譯參數

```
$(TARGET): $(SRCS)
→$(CC) -o $@ $(CFLAGS) $^
```

展開變數，依展開的變數編譯檔案

```
clean:
→rm -f $(TARGET)
```

清除產生的檔案規則

```
$ ls
hello.c  Makefile
$ make
cc -o hello -g -Wall -Werror hello.c
$ ls
hello hello.c  Makefile
$ make clean
rm -f hello
$ make
cc -o hello -g -Wall -Werror hello.c
```

目錄下有原始檔和Makefile

編譯有展開變數並指定對應檔案目錄下有原始檔和Makefile

執行檔已經產生

清除執行檔並重新編譯

條件判斷範例

LOGNAME是環境變數!

```
if [ $(LOGNAME) = test ]
    $(warning This is test)
else
    $(warning $(LOGNAME))
endif

fake:
    @echo
```

LOGNAME是環境變數!

```
$ export |grep LOGNAME  
declare -x LOGNAME="test"  
$ make  
Makefile:2: This is test  
  
$ make LOGNAME=fake  
Makefile:4: fake
```

人肉設定LOGNAME內容

function節錄

- 語法
 - \$(函數名稱 參數)

分類	函數名稱	說明	範例 (請貼到Makefile實測!)
訊息	\$(warning 訊息)	顯示警告訊息以及對應的行號	\$(warning Your gcc version is too old)
	\$(error 訊息)	顯示錯誤訊息、對應的行號後結束 make	conf=my_file \$(error file \$(conf) not found)
字串處理	\$(subst from,to,處理文字)	字串替換,後面空白為參數的一部份	\$(warning \$(subst .c,.o,test.c hello.c))
	\$(patsubst pattern,替換文字,處理文字)	pattern字串替換,後面空白為參數的一部份。 %代表任意長度的任意字元。	\$(warning \$(patsubst t%.c,a%.o,test.c hello.c))
其他	\$(shell 命令)	執行命令, 回傳文字結果	\$(warning \$(shell ls /))

連續技

- OBJS=\$(patsubst %.c,%.o,\$(shell ls *.c))


常見錯誤：每個recipe 執行狀態不延續 造成的悲劇

範例：建立一個目錄，進入該目錄前後
印出目前工作目錄確認切換目錄成功

```
fake:  
——>pwd  
——>mkdir -p test  
——>cd test  
——>pwd
```

開獎，沒有切換到test目錄

```
$ make  
pwd  
/tmp/temp  
mkdir -p test  
cd test  
pwd  
/tmp/temp
```



窄宅看的, 可以看到make處理recipe的方式是產生新的process, 執行recipe, 然後結束該process

```
$ strace -f make
```

```
vfork(Process 8949 attached (waiting for parent)
```

```
Process 8949 resumed (parent 8947 ready)
```

```
) = 8949
```

```
[pid 8949] execve("/bin/mkdir", ["mkdir", "-p", "test"],  
[/* 42 vars */]) = 0
```

```
<... wait4 resumed> [{WIFEXITED(s) &&  
WEXITSTATUS(s) == 0}], 0, NULL) = 8949
```

```
--- SIGCHLD (Child exited) @ 0 (0) ---
```

小結

- GNU make: 協助編譯的時候決定
 - 那部份要重編
 - 指定重新編譯的動作
- 所以寫Makefile, 主要描述
 - 產生檔案和原始檔的關聯性
 - 當這些檔案更動時間關係有變化的時候, 該做什麼事?

補充1: 沒有Makefile的make

目前目錄沒有Makefile

```
$ ls
```

```
hello.c
```

照樣make嘿嘿

```
$ make hello.o
```

```
cc -c -o hello.o hello.c
```

```
$ make hello
```

照樣make++

```
cc hello.o -o hello
```

Implicit Rules

```
$ make --help
Usage: make [options] [target] ...
Options:
...
-C DIRECTORY, --directory=DIRECTORY
                        Change to DIRECTORY before doing anything.
-d
                        Print lots of debugging information.
...
-n, --just-print, --dry-run, --recon
                        Don't actually run any commands; just print them.
...
-p, --print-data-base
                        Print make's internal database.
```

三小？

make -p

```
$ make -p
...
CC = cc
CPP = $(CC) -E
LINK.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
LINK.o = $(CC) $(LDFLAGS) $(TARGET_ARCH)
...
```

CPP這邊可不是C++, 有
興趣man cpp

還記得變數嗎？

make -p (接關)

```
...
%: %.o
» $(LINK.o) ^ $(LOADLIBES) $(LDLIBS) -o $@
%: %.c
» $(LINK.c) ^ $(LOADLIBES) $(LDLIBS) -o $@
%.o: %.c
» $(COMPILE.c) $(OUTPUT_OPTION) $<
.o:
» $(LINK.o) ^ $(LOADLIBES) $(LDLIBS) -o $@
.c.o:
» $(COMPILE.c) $(OUTPUT_OPTION) $<
.c:
» $(LINK.c) ^ $(LOADLIBES) $(LDLIBS) -o $@
```

make有內建預設的處理規則，請和上頁變數以及前面的語法對照

補充2: .PHONY

範例Makefile, 包含產生binary和清除binary

```
TARGET=hello

$(TARGET): $(TARGET).c
——>$(CC) -o $@ $^

clean:
——>rm -f $(TARGET)
```

玩看看

清除binary

產生新檔, 名稱為clean

檔案clean存在, 無法清除binary

必須刪除檔案clean才能執行target

```
$ make clean  
rm -f hello  
$ touch clean  
$ make clean  
make: `clean' is up to date.  
$ rm clean  
$ make clean  
rm -f hello
```


分析

- recipe 的執行和target以及prerequisites檔案時間資訊有關
- clean的目的並不是要產生clean檔案，也就是說**target執行recipe和target是否為檔案無關**

.PHONY

- 和檔案無關的target
- 用法
.PHONY: 以空白隔開的TARGET名稱
- 範例
.PHONY: clean install

修正

.PHONY在這邊

```
.PHONY: clean
```

```
TARGET=hello
```

```
$(TARGET): $(TARGET).c
```

```
→ $(CC) -o $@ $^
```

```
clean:
```

```
→ rm -f $(TARGET)
```

看看效果

```
$ make clean  
rm -f hello  
$ touch clean  
$ make clean  
rm -f hello
```

延伸題材/回家功課

- 如何自動進入不同目錄Make?
- 如何自動產生C source檔和Header檔案rule?
 - hello.c includes f1.h, f2.h, 兩天後又加入f3.h。手動改很累。
- 有沒有辦法把所有的設定放在檔案內給Makefile include?

剩下就是細節，請自行
看書，規劃設計自己的
實習課。

Outline

- GNU Make
- Autotools
- CMake
- 參考資料
- Q & A

複習: GNU Make小結

- GNU make: 協助編譯的時候決定
 - 那部份要重編
 - 指定重新編譯的動作
- 所以寫Makefile, 主要描述
 - 產生檔案和原始檔的關聯性
 - 當這些檔案更動時間關係有變化的時候, 該做什麼事?

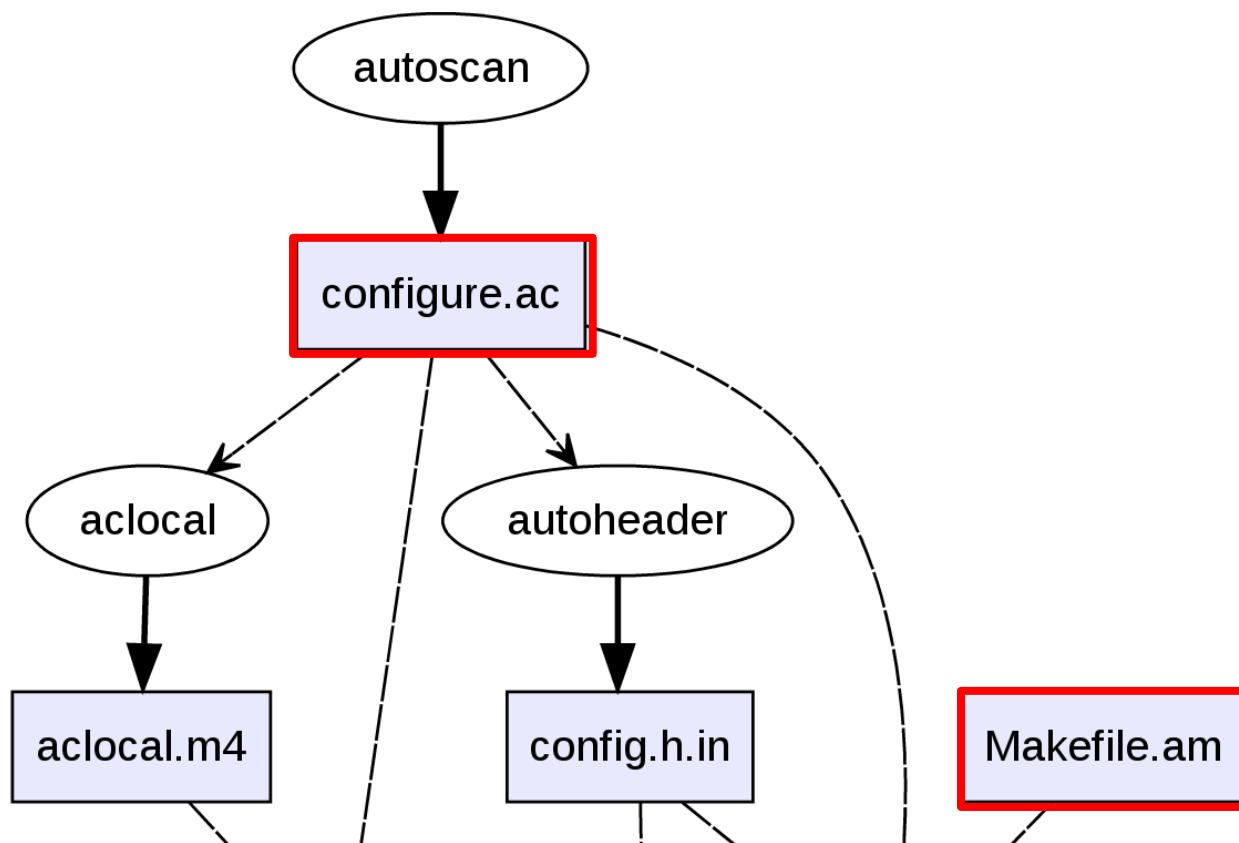
那麼會什麼還要有autotools

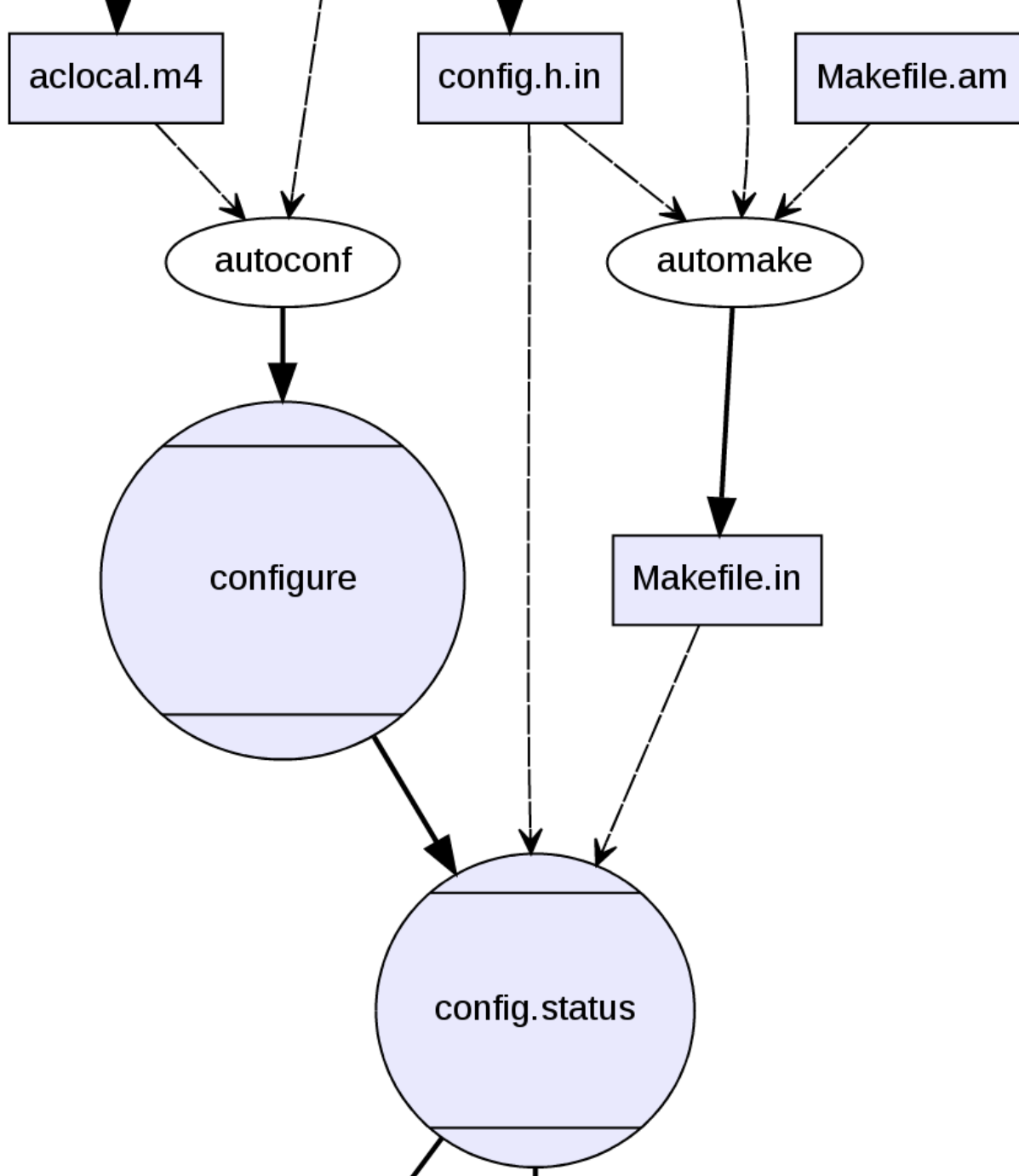
- 跨平台的問題
 - memset v.s. bzero
 - 路徑、檔案不同
 - system call不同
- 同平台
 - 函式庫版本不同, prototype可能不同
- 相依性問題

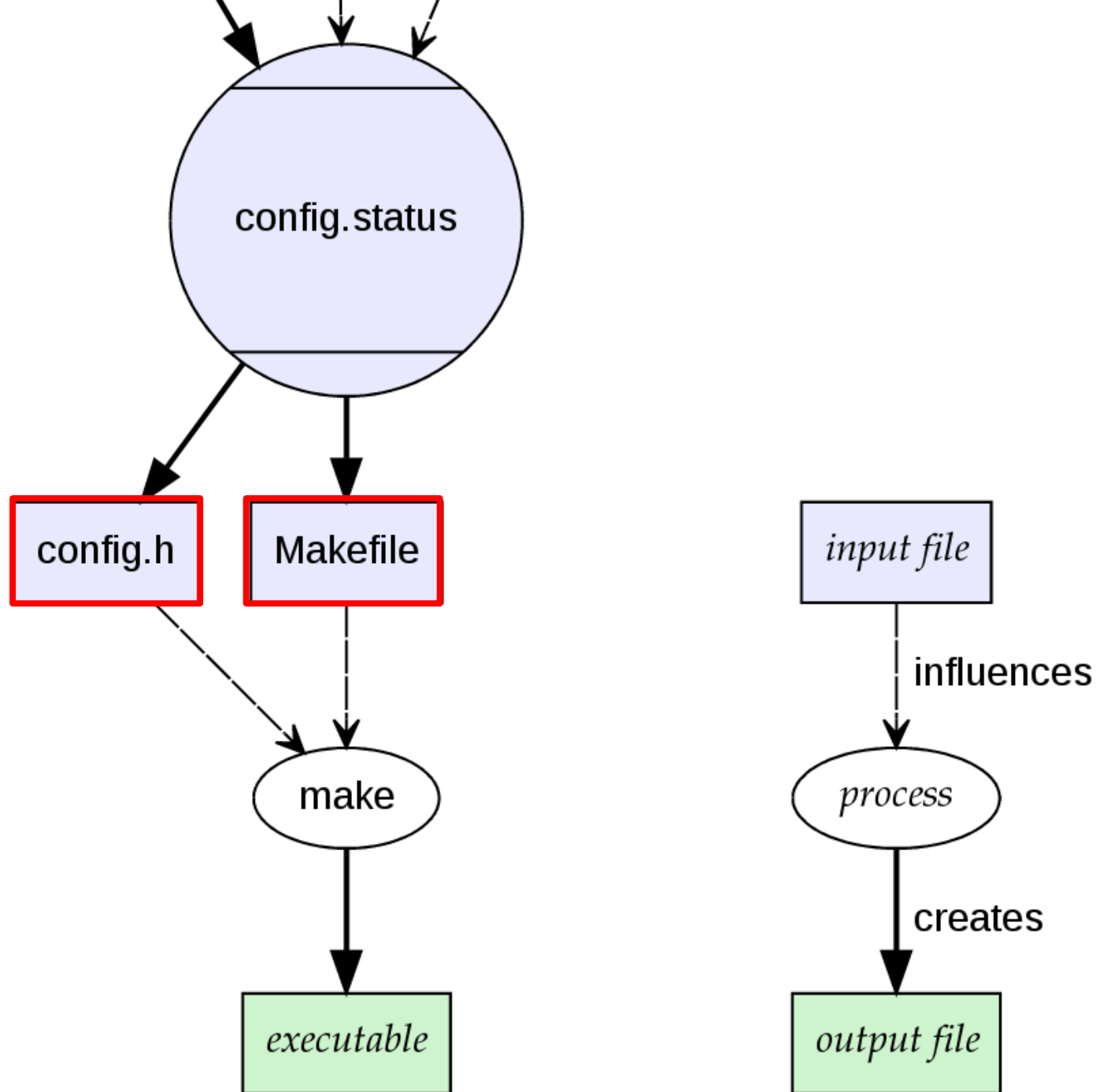
autotools

- autotool的目的就是產生平台上可以編譯的環境
- 為了達到這樣的目的，系統需要做到下面的功能
 - 檢查平台環境
 - 產生Makefile

流程







很複雜？口頭講就是

1. 自幹或跑工具後產生更動configure.ac, 和寫Makefile.am
2. 跑工具產生configure, Makefile.in和config.in
3. 跑configure, 產生Makefile和config.h
4. make; make install

還不懂？來看範例

```
$ tree
.
├── autogen.sh
├── configure.ac
├── include
│   ├── liba.h
│   └── libb.h
├── libs
│   ├── liba.c
│   ├── libb.c
│   └── Makefile.am
├── Makefile.am
└── src
    ├── Makefile.am
    └── test.c

3 directories, 10 files
```

你要寫的東西

autoreconf幫你呼叫相關工具如aclocal, autoconf, automake等

```
$ cat autogen.sh
#!/bin/sh
autoreconf --install
$ ./autogen.sh
libtoolize: Consider adding 'AC_CONFIG_MACRO_
DIR([m4])' to configure.ac and
libtoolize: rerunning libtoolize, to keep the
correct libtool macros in-tree.
libtoolize: Consider adding '-I m4' to ACLOCAL_
AMFLAGS in Makefile.am.
```



```
$ tree
```

```
├── aclocal.m4
├── autogen.sh
├── autom4te.cache
│   ├── output.0
│   ├── output.1
│   ├── requests
│   ├── traces.0
│   └── traces.1
├── config.guess
├── config.h.in
├── config.sub
├── configure
├── configure.ac
├── depcomp
├── include
│   ├── liba.h
│   └── libb.h
└── install-sh
```

產生出來的檔案

```
install-sh
libs
  liba.c
  libb.c
  Makefile.am
  Makefile.in
ltmain.sh
Makefile.am
Makefile.in
missing
src
  Makefile.am
  Makefile.in
  test.c

4 directories, 27 files
```

產生出來的檔案

configure.ac

要求版本

AC_PREREQ([2.68])

套件資訊

AC_INIT([Test_Autotools], [0], [test])

給Automake資訊, foreign表示不用GNU標準

也就是不需要changelog, AUTHORS等檔案

AM_INIT_AUTOMAKE([**foreign** -Wall -Werror])

configure.ac

config檔案

AC_CONFIG_HEADERS([config.h])

本次demo使用 static library

AC_PROG_RANLIB

Makefile 路徑

AC_CONFIG_FILES([Makefile src/Makefile
libs/Makefile])

configure.ac

搜尋CC 編譯器

AC_PROG_CC

結束config, 開始產生相關檔案

AC_OUTPUT

Makfile.am

```
SUBDIRS = libs src
```

libs/Makefile.am

指定include路徑

AM_CFLAGS = -I../include

產生liba.a 和libb.b

lib_代表安裝時要放在\$(prefix)/lib中

預設prefix=/usr/local

lib_LIBRARIES = liba.a libb.a

libs/Makefile.am

產生liba.a的相依檔案

liba_a_SOURCES = liba.c liba.h

產生libb.a的相依檔案

libb_a_SOURCES = libb.c libb.h

安裝到\$(prefix)/include的檔案

include_HEADERS = ../include/liba.h ..
/include/libb.h

src/Makefile.am

指定link哪些library

LDADD = ../libs/liba.a ../libs/libb.a

指定include路徑

AM_CFLAGS = -I../include

src/Makefile.am

安裝到\$(prefix)/bin

bin_PROGRAMS = test

產生的檔案相依的檔案

test_SOURCES = test.c

- configure
- make
- make install
- make dist # 自動幫你打包tarball

```
$ ./configure --prefix=`pwd`/test
checking for a BSD-compatible install...
/usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir
-p
checking for gawk... gawk
...
checking for GNU libc compatible malloc... yes
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating libs/Makefile
config.status: creating config.h
config.status: executing depfiles commands
```

```
$ make
make all-recursive
make[1]: Entering directory `/tmp/temp'
Making all in libs
make[2]: Entering directory `/tmp/temp/libs'
gcc -DHAVE_CONFIG_H -I. -I.. -I../include -g -O2 -MT liba.o -MD -MP -MF .deps/liba.Tpo -c -o liba.o liba.c
mv -f .deps/liba.Tpo .deps/liba.Po
rm -f liba.a
ar cru liba.a liba.o
...
gcc -I../include -g -O2 -o test test.o ../libs/liba.a ../libs/libb.a
make[2]: Leaving directory `/tmp/temp/src'
make[2]: Entering directory `/tmp/temp'
make[2]: Leaving directory `/tmp/temp'
make[1]: Leaving directory `/tmp/temp'
```

```
$ make install
Making install in libs
make[1]: Entering directory `/tmp/temp/libs'
make[2]: Entering directory `/tmp/temp/libs'
test -z "/tmp/temp/test/lib" || /bin/mkdir -p "/tm
p/temp/test/lib"
  /usr/bin/install -c -m 644 liba.a libb.a '/tmp/t
emp/test/lib'
  ( cd '/tmp/temp/test/lib' && ranlib liba.a )
  ( cd '/tmp/temp/test/lib' && ranlib libb.a )
test -z "/tmp/temp/test/include" || /bin/mkdir -p
"/tmp/temp/test/include"
  /usr/bin/install -c -m 644 ../include/liba.h ../i
nclude/libb.h '/tmp/temp/test/include'
make[2]: Leaving directory `/tmp/temp/libs'
make[1]: Leaving directory `/tmp/temp/libs'
Making install in src
make[1]: Entering directory `/tmp/temp/src'
make[2]: Entering directory `/tmp/temp/src'
test -z "/tmp/temp/test/bin" || /bin/mkdir -p "/tm
p/temp/test/bin"
  /usr/bin/install -c test '/tmp/temp/test/bin'
make[2]: Nothing to be done for `install-data-am'.
make[2]: Leaving directory `/tmp/temp/src'
make[1]: Leaving directory `/tmp/temp/src'
make[1]: Entering directory `/tmp/temp'
make[2]: Entering directory `/tmp/temp'
make[2]: Nothing to be done for `install-exec-am'.
make[2]: Nothing to be done for `install-data-am'.
make[2]: Leaving directory `/tmp/temp'
make[1]: Leaving directory `/tmp/temp'
```

```
$ tree test
```

```
test
```

```
├── bin
│   └── test
├── include
│   ├── liba.h
│   └── libb.h
└── lib
    ├── liba.a
    └── libb.a
```

```
3 directories, 5 files
```

小結

- 使用autotools需要自己寫
 - configure.ac
 - Makefile.am
- 上面的範例極度簡略

延伸題材/回家功課

- 如何產生shared library?
- 是否有更聰明的library產生方式?
 - hint: libtools

剩下就是細節，請自行
看書，規劃設計自己的
實習課。

Outline

- GNU Make
- Autotools
- **CMake**
- 參考資料
- Q & A

關於CMake

CMake是1999年推出的開源自由軟體計畫，目的是提供不同平台之間共同的編譯環境。

特性：

- 支援不同平台
- 可以將Build和原本程式碼分開
 - out-place build
 - in-place build
- 支援cache加快編譯速度

流程

- 撰寫**CMakeLists.txt**
- 使用者執行cmake
 - 產生該平台對應的編譯環境檔案如Makefile等
- 使用者執行平台上的編譯方法
 - 如make
- 使用者執行cmake install安裝軟體。

範例

```
$ tree
.
├── CMakeLists.txt
├── include
│   ├── CMakeLists.txt
│   ├── liba.h
│   └── libb.h
├── libs
│   ├── CMakeLists.txt
│   ├── liba.c
│   └── libb.c
├── readme.txt
└── src
    ├── CMakeLists.txt
    └── test.c

3 directories, 10 files
```

你要寫的東西

The diagram consists of four orange arrows originating from a central text box on the right and pointing to four specific files in the directory tree on the left. The files are: 1. The root CMakeLists.txt file. 2. The CMakeLists.txt file inside the 'include' directory. 3. The CMakeLists.txt file inside the 'libs' directory. 4. The CMakeLists.txt file inside the 'src' directory. Each of these four files is highlighted with a red rectangular border in the terminal output.

CMakeLists.txt

版本需求

```
cmake_minimum_required(VERSION 2.8)
```

你的Project名稱

```
project(testcmake)
```

CMakeLists.txt

設定變數

set(**SRC_DIR** src)

set(**LIB_DIR** libs)

set(**INC_DIR** include)

Compile flags

set(CMAKE_C_FLAGS "-Wall -Werror")

CMakeLists.txt

指令include目錄

```
include_directories(${INC_DIR})
```

告訴CMake要去下列的目錄編譯

```
add_subdirectory(${SRC_DIR})
```

```
add_subdirectory(${LIB_DIR})
```

```
add_subdirectory(${INC_DIR})
```

libs/CMakeLists.txt

設定變數, 指定library相依於哪個檔案

```
set(liba_SRCS liba.c)
```

```
set(libb_SRCS libb.c)
```

指定編譯型式為shared library

```
add_library(a SHARED ${liba_SRCS})
```

```
add_library(b SHARED ${libb_SRCS})
```

libs/CMakeLists.txt

安裝格式

install(TARGETS 函式庫名稱 LIBRARY
DESTINATION 安裝目錄路徑)

install(TARGETS **a b** LIBRARY
 DESTINATION **lib**)

src/CMakeLists.txt

設定變數

```
set(test_SRCS test.c)
```

產生執行檔

```
add_executable(${PROJECT_NAME}  
               ${test_SRCS})
```

指令link函式庫

```
target_link_libraries(${PROJECT_NAME} a b)
```

src/CMakeLists.txt

安裝格式

install(TARGETS 執行檔名稱

DESTINATION 安裝目錄路徑)

install(TARGETS \${PROJECT_NAME}
 DESTINATION **bin**)

include/CMakeLists.txt

```
# install(FILES Header檔名稱  
#          DESTINATION 安裝目錄路徑)  
install(FILES liba.h libb.h  
          DESTINATION include)
```

out-place build

```
$ mkdir build
$ cd build/
$ cmake ../ -DCMAKE_INSTALL_PREFIX=`pwd`/test
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc --
   works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++
-- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/tmp/build
```



```
$ make
Scanning dependencies of target b
[ 33%] Building C object libs/CMakeFiles/b.dir/libb.c.o
Linking C shared library libb.so
[ 33%] Built target b
Scanning dependencies of target a
[ 66%] Building C object libs/CMakeFiles/a.dir/liba.c.o
Linking C shared library liba.so
[ 66%] Built target a
Scanning dependencies of target testcmake
[100%] Building C object src/CMakeFiles/testcmake.dir/test.c.o
Linking C executable testcmake
[100%] Built target testcmake
```

```
$ make install
[ 33%] Built target b
[ 66%] Built target a
[100%] Built target testcmake
Install the project...
-- Install configuration: "Debug"
-- Installing: /tmp/tmp/build/test/bin/testcmake
-- Removed runtime path from "/tmp/tmp/build/test/
bin/testcmake"
-- Installing: /tmp/tmp/build/test/lib/liba.so
-- Installing: /tmp/tmp/build/test/lib/libb.so
-- Installing: /tmp/tmp/build/test/include/liba.h
-- Installing: /tmp/tmp/build/test/include/libb.h
```

```
$ tree test/  
test/  
├── bin  
│   └── testcmake  
├── include  
│   ├── liba.h  
│   └── libb.h  
└── lib  
    ├── liba.so  
    └── libb.so
```

```
3 directories, 5 files
```

小結

- 你要寫CMakeLists.txt, 指定要進入哪些目錄編譯, 或是相關的檔案以及預期編譯和安裝的型態
- 上面的範例極度簡略

延伸題材/回家功課

- 詳細語法
 - 變數設定
 - 條件設定
 - 巨集和函數
- 打包方式

剩下就是細節，請自行
看書，規劃設計自己的
實習課。

Outline

- GNU Make
- Autotools
- CMake
- 參考資料
- Q & A

參考資料

- GNU Make手冊
 - <http://www.gnu.org/software/make/manual/make.html>
- GNU Make 快速參考
 - <http://www.gnu.org/software/make/manual/make.html#Quick-Reference>
- GNU Automake手冊
 - <http://www.gnu.org/software/automake/manual/automake.html>
- Alexandre Duret Lutz: Autotools Tutorial (大推)
 - <https://www.lrde.epita.fr/~adl/autotools.html>

參考資料

- CMake Wiki
 - <http://www.cmake.org/Wiki/CMake>
- CMake-tutorial (大推)
 - <https://github.com/TheErk/CMake-tutorial>

Outline

- GNU Make
- Autotools
- CMake
- 參考資料
- Q & A