Manteniment de projectes: make (2)

Programació de Baix Nivell

Sebastià Vila-Marta

Enginyeria de Sistemes TIC
Universitat Politècnica de Catalunya
http://epsem.upc.edu

1 d'abril de 2020

Variables

- En un makefile es poden usar variables.
- Típicament s'usen per a estructurar el makefile de manera que sigui més fàcil el seu manteniment.
- Les variables són de tipus `string`,
- Tenen sintaxis diferenciades quan actuen de l-expr o r-exp. Si volem referir-nos a la variable ho fem amb el seu identificador. D'altra banda, si ens volem referir al valor que contenen ho fem amb dòlar parèntesi:

```
COMPILADOR     $(COMPILADOR)
```

- L'assignació de variables és denota amb =

```
COMPILADOR = gcc
OPCIONS = -Wall
COMP = $(COMPILADOR) $(OPCIONS)
```

- Noteu que generalment no són necessàries les cometes pels strings i que els espais de davant i darrera de l'string s'ignoren.

Variables: exemple d'ús

- Un ús típic podria ser:

```
COMP = gcc -c

datafile.o: datafile.c datafile.h queue.h person.h
    $(COMP) datafile.c
person.o: person.c person.h
    $(COMP) person.c
queue.o: queue.c queue.h
    $(COMP) queue.c
```

- Si ara volem variar les opcions de compilació:

```
COMP = gcc -c -std=c99 -Wall

datafile.o: datafile.c datafile.h queue.h person.h
    $(COMP) datafile.c
person.o: person.c person.h
    $(COMP) person.c
queue.o: queue.c queue.h
    $(COMP) queue.c
```

Variables automàtiques

- Són variables que prenen un valor automàticament per a cada regla que s'executa.
- Només poden usar-se en la part de l'acció d'una regla.
- Les més habituals són:

`$@`	El target de la regla
`$<`	La primera dependència de la regla
`$^`	La llista de totes les dependències
`$?`	Les dependències més modernes que el target

- En la regla següent:

```
stack.o: stack.c stack.h
```

La variable `$@` val «`stack.o`»; `$<` val «`stack.c`» i `$^` val la llista de noms de fitxer «`stack.c stack.h`».

Variables automàtiques (2)

- Les variables automàtiques simplifiquen l'escriptura de regles:

```
main: modul1.o modul2.o modul3.o modul4.o
    gcc $^ -o $@
```

- Les podem combinar amb les variables ordinàries per crear «accions genèriques», per exemple:

```
COMP = gcc -c $<
LINK = gcc $^ -o $@

main: main.o datafile.o person.o stack.o queue.o
    $(LINK)
stack.o: stack.c stack.h
    $(COMP)
datafile.o: datafile.c datafile.h queue.h person.h
    $(COMP)
person.o: person.c person.h
    $(COMP)
queue.o: queue.c queue.h
    $(COMP)
main.o: main.c
    $(COMP)
```

Regles predefinides

- Make té un conjunt de regles predefinides que abasten els casos més corrents.
- El seu us simplifica moltíssim l'escriptura de Makefiles.
- Les regles estan definides en base a unes veriables ben establertes: el valor de les variables permet «configurar» l'aplicació de les regles.
- Les que més ens afecten són:

compilació C	`$(CC) $(CPPFLAGS) $(CFLAGS) -c <font>`
link	`$(CC) $(LDFLAGS) <objs> $(LOADLIBES) $(LDLIBS)`

- Dedueixen automàticament la dependència principal. Així, la regla:

```
main: datafile.o person.o stack.o queue.o
```

indica —tot i no explicitar-ho— que `main.o` també és una dependència.

Regles pre-definides (2)

- El Makefile de l'exemple usant regles predefinides seria:

```
CC = gcc
CFLAGS = -Wall

main: datafile.o person.o stack.o queue.o
stack.o: stack.h
datafile.o: datafile.h queue.h person.h
person.o: person.h
queue.o: queue.h
```

- Noteu que:
  - Han desaparegut les accions de les regles esperant que s'adoptin les accions predefinides.
  - Han desaparegut les dependències principals de les regles: les aporten automàticament les regles predefinides.
  - S'han assignat els valors escaients a les variables `CC` i `CFLAGS` per tal que les ordres emeses per les regles predefinides siguin les escaients.

Regles genèriques

- Sovint seria útil tenir regles genèriques que no troben entre les regles predefinides. Per ex: per traduïr fitxers `.rst` a `pdf`.
- En aquest casos es poden definir regles genèriques en el mateix `Makefile`.
- Exemple:

```
%.pdf: %.rst
    rst2pdf $(RSTPDFFLAGS) -o $@ $<
```

- Si amb la regla generica definida li demanem a `make` que actualitzti `exemple.pdf`, deduirà que depèn de `exemple.rst` i, si és necessari, l'actualizarà executant l'ordre
  `rst2pdf -o exemple.pdf exemple.rst`
- Si la variable `RSTPDFFLAGS` tingués el valor «`-l ca`», llavors l'ordre executada seria `rst2pdf -l ca -o exemple.pdf exemple.rst`

Targets «phony»

- Un target «phony» (fals) és un target que no correspon a cap fitxer.
- Els target «phony» només entren en joc quan es demana explícitament actualitzar-los.
- La sintaxi per declarar que un target és «phony» és la següent:

```
.PHONY: <target phony 1>, <target phony 2>, ...
```

- Exemple: un projecte té tres executables i volem que una invocació sense arguments de `make` els actualitzi tots tres.

```
.PHONY: all

all: exec1 exec2 exec3
exec1: exec1.o ...
...
```

Com ara «all» és el primer target del `Makefile`, sempre serà l'objectiu si s'executa `make` sense arguments.

Usos col·laterals I

- A banda de l'ús central com a eina per construir un projecte, la funcionalitat de `make` s'usa per alleugerir altres tasques habituals.
- Sovint es fa amb combinació amb els targets phony.
- Observeu el següent makefile:

```
.PHONY: clean veryclean

clean:
    \rm -f *~ *.o

veryclean: clean
    \rm -f main *.pdf
```

- Què succeirà si executem

```
$ make clean
```

Fixeu-vos que es «farà neteja» de fitxers intermedis del projecte: backups d'emacs, objectes, etc.

Usos col·laterals II

- `veryclean` faria primer un `clean` i esborraria també els executables, pdf's i altres fitxers «calculables».
- Un altre exemple d'ús col·lateral podria ser aquest:

```
pract-6-equip-8.tar.gz: README main.c stack.c \
                        stack.h mod.c mod.h
    tar zcvf $@ $^
```

  Observeu que us serviria per crear el tarfile que heu de lliurar sense haver de patir i/o recordar què cal afegir-hi, quan cal fer-ho i com.

  Noteu la manera de tallar una línia massa llarga en la regla.

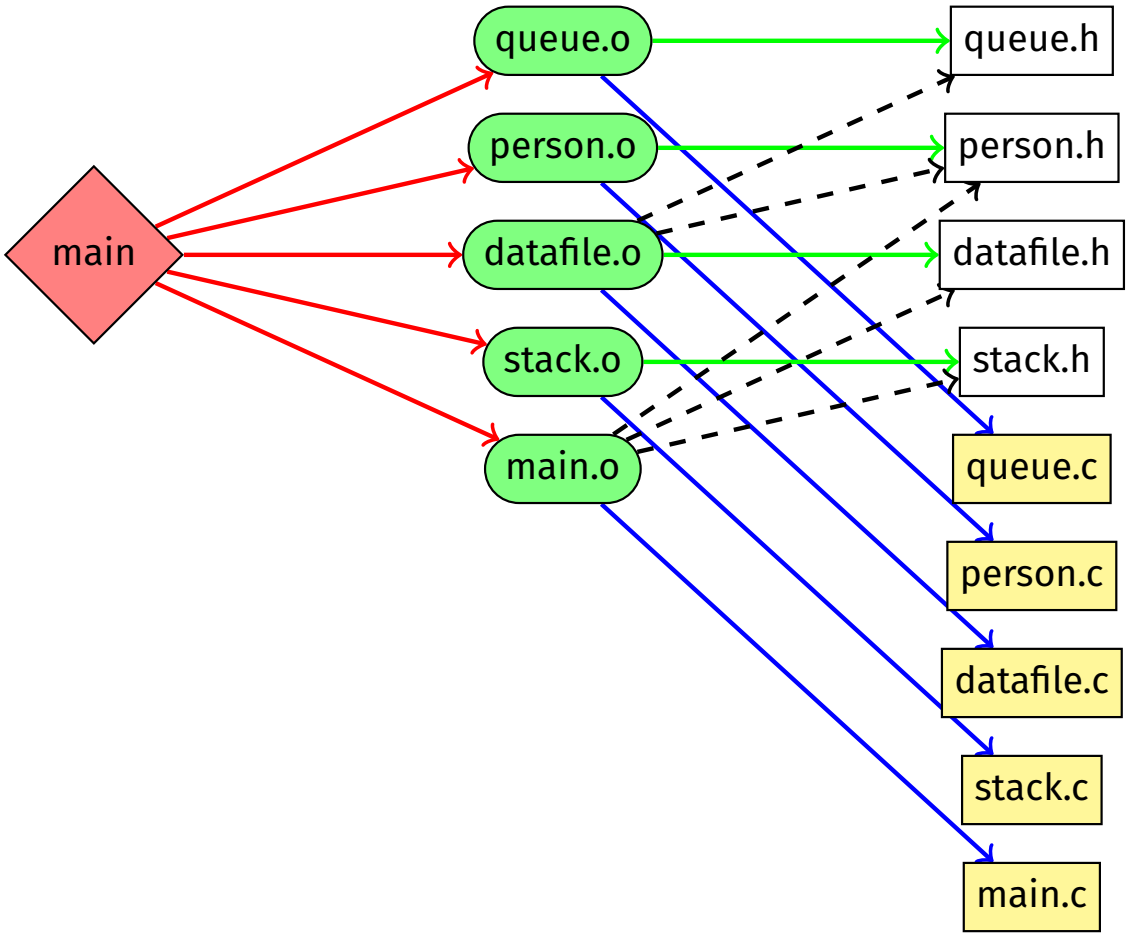
Obtenció automàtica de les dependències

- Calcular les dependències entre fitxers d'un programa és una feina delicada i feixuga. Les errades provoquen errades de construcció que són especialment empipadores.
- El compilador de C de GNU sap calcular les dependències i escriure-les en format Makefile.
- Passeu-li al compilador els fonts c del vostre projecte, i us escriurà en el canal de sortida la llista de dependències. Per exemple:

```
$ gcc -MM *.c
mtbl.o: mtbl.c mchar.h mtbl.h
queue.o: queue.c queue.h
semaph.o: semaph.c semaph.h lamp.h
serial.o: serial.c queue.h serial.h
```

Un Makefile «full equiped»

- Recordem l'estructura del projecte exemple usat a la sessió anterior:



- Anem a construir un `Makefile` complet per aquest projecte. Suposarem que es tracta d'una pràctica com les de l'assignatura.

Els detalls del Makefile I

- Organitzarem el `Makefile` parametritzat en base a variables.
- Usarem regles predefinides.
- El complementarem amb algunes funcionalitats colaterals que facilitin la gestió del projecte.
- Les dependències (deduides directament del graf) són:

```
main: datafile.o person.o stack.o queue.o
stack.o: stack.h
datafile.o: datafile.h queue.h person.h
person.o: person.h
queue.o: queue.h
```

  Noteu que les dependències principals no hi són per que les dedueixen les regles automàtiques. Això fa que el mòdul `main` ni tant sols hi surti.

- Les variables que cal configurar per tal que les regles automàtiques facin la feina correcta són:

```
CC = gcc
CFLAGS = -std=c99 -Wall
```

Els detalls del Makefile II

- També definirem algunes variables més per configurar quins mòduls constitueixen el projecte, i la identificació de la pràctica i l'equip:

```
NPRACT = 9
NEQUIP = 4
MODULS = datafile person stack queue
MODULPRINC = main
```

- De les variables anteriors definirem algunes variables intermèdies (no configurables) que poden fer servei:

```
TARNAME = pract-$(NPRACT)-equip-$(NEQUIP)
```

- Afegirem targets específics per netejar el directori com hem explicat abans:

```
.PHONY: clean veryclean

clean:
    \rm -f *~ *.o

veryclean: clean
    \rm -f main $(TARNAME).tar.gz
```

Els detalls del Makefile III

- Afegirem un target per fer el procés de *release* —la preparació d'una versió pel client–.

```
.PHONY: release

release: $(TARNAME).tar.gz

$(TARNAME).tar.gz: README \
                   $(addsuffix .c, $(MODULS) $(MODULPRINC)) \
                   $(addsuffix .h, $(MODULS))
    tar zcvf $@ $^
```

  Noteu que hem usat una funció interna de GNU make (`addsuffix`) que permet manipular noms de fitxer. Consulteu el manual de GNU make!!

- Finalment, el `Makefile` amb algun detall cosmètic afegit esdevé:

Els detalls del Makefile IV

```
CC = gcc
CFLAGS = -std=c99 -Wall
NPRACT = 9
NEQUIP = 4
MODULS = datafile person stack queue
MODULPRINC = main

TARNAME = pract-$(NPRACT)-equip-$(NEQUIP)

.PHONY: clean veryclean release all

all: main

clean:
    \rm -f *~ *.o

veryclean: clean
    \rm -f main $(TARNAME).tar.gz

release: $(TARNAME).tar.gz
```

Els detalls del Makefile V

```
$(TARNAME).tar.gz: README \
                   $(addsuffix .c, $(MODULS) $(MODULPRINC)) \
                   $(addsuffix .h, $(MODULS))
    tar zcvf $@ $^

main: $(addsuffix .o, $(MODULS))
stack.o: stack.h
datafile.o: datafile.h queue.h person.h
person.o: person.h
queue.o: queue.h
```