Curs 20-21

UD 4. Eines d'automatització

L'eina make

Entorns de desenvolupament

IES Jaume II El Just
Tavernes de la Valldigna

Continguts

Make i makefile

L'eina GNU Make ens permet agilitzar la tasca de compilar codi des de la terminal, i ens evita escriure les ordre de compilació cada vegada.

Make requereix d'uns fitxers, anomenats *makefiles* que són fitxers que s'inclouen en la carpeta arrel d'un projecte, i que li indiquen a *make* què fer amb cadascun dels fitxers de codi per tal de compilar-los. Aquests fitxers, podem dir que es tracta de *receptes* que indiquen *com* generar els executables.

Si no tenim *make* instal·lat al nostre sistema, ho podem fer amb:

```
$ sudo apt install make
```

Compilant un «Hola Mundo»

Si tenim un programa senzill en C, de l'estil «Hola Món»:

```c
#include <stdio.h>

int main()
{
        printf("Hola món\n");
        return 0;
}
```

Recordem que per compilar aquest fitxer, podem escriure:

```
gcc hola.c
```

Amb el que obtindrem un fitxer executable anomenat `a.out`. Si volem que el fitxer d'eixida tinga altre nom, ho podem especificat amb el paràmetre `-o`:

```
gcc hola.c -o hola
```

Tingueu en compte que podem utilitzar tant `gcc` com `cc`.

`gcc` a més, admet altres arguments, entre els que podem trobar `-Wall` i `-g`, que serveixen per mostrar també tots els `Warnings` durant la compilació:

```
gcc -Wall -g hola.c -o hola
```

Utilitzant make per a l'«Hola Mundo»

Ara anem a veure com utilitzar el `make` amb la mateixa finalitat que `gcc` o `cc`. Si ens situem a la mateixa carpeta i llancem `make hola`, obtindrem:

```
$ make hola
cc     hola.c   -o hola
```

Com veiem, make interpreta que volem crear l'executable `hola` i enten que el fitxer font es dirà `hola.c`.

> **TO-DO**
>
> - Proveu a llançar les ordres anteriors sobre el fitxer `hola.c`.
> - Comproveu el resultat executant els fitxers compilats

Funcions, llibreríes i fitxers de capçalera

Anem a avançar una miqueta més amb el funcioanment de make. Tenim el següent fitxer de codi en C, anomenat `calc.c`:

```c
#include <stdio.h>

int suma(int op1, int op2){
    return (op1+op2);
}

int resta(int op1, int op2){
    return (op1-op2);
}

int multiplica(int op1, int op2){
    return (op1*op2);
}

int divideix(int op1, int op2){
    return (op1/op2);
}

int main()
{
        int a=10;
        int b=5;

        printf("La suma de %d i %d és %d\n", a, b, suma(a,b));
        printf("La resta entre %d i %d és %d\n", a, b, resta(a,b));
        printf("La multiplicació de %d i %d és %d\n", a, b, multiplica(
            a,b));
        printf("La divisió entre %d i %d és %d\n", a, b, divideix(a,b))
            ;
```

```
28 }
```

Aquest fitxer defineix quatre **funcions**: *suma*, *resta*, *multiplica* i *divideix* (a banda de la funció principal *main*). Veiem què vol dir açò.

- Les **funcions** en els llenguatges de programació imperatius, són **subprogrames** que reben unes dades, fan operacions amb elles, i ens retornen un resultat.
- Aquests dades que reben les funcions s'anomenen **arguments**, i s'especifiquen entre parèntesi a la dreta del nom de la funció, indicant el tipus de dada de què es tracta, i el nom de la variable com volem que s'anomene dins aquesta funció.
- Si es fixeu, a l'esquerra del nom de la funció s'indica el **tipus** de la informació que cada funció torna, i la última ordre de la funció és un `return` amb què especifiquem el valor de retorn.
- Per utilitzar una funció, s'ha d'***invocar*** aquesta mitjançant el seu nom, i passar-li els valors que volem que prenguen els arguments, bé directament o be mitjançant variables. Aquests arguments són posicionals, és a dir, l'ordre en què els indiquem en la crida és el mateix en què s'agafen en la funció. Evidentment, els arguments que posem tant en la crida com en la definició de la funció han de coincidir en nombre, i també en el tipus.

Com veiem, hem definit quatre funcions que implementen senzilles operacions matemàtiques, que també podriem haver realitzat sense definir cap funció. Les funcions normalment tenen un codi més extens i realitzen tasques més complexes. La finalitat de les funcions és ajudar-nos a fer un codi més *modular*, i que algunes parts es puguen reutilitzar.

Anant un pas més enllà, tenim el que serien les llibreríes. Un fitxer de llibrería, no és més que un fitxer que conté certes funcions d'utilitat. Les llibreríes poden ser del sistema o creades per nosaltres per al nostre projecte. Sense anar més lluny, quan indiquem `include <stdio.h>`, el que fem és *carregar* la capçalera de la llibrería `stdio.h`, que ens proporciona funcions com `printf` per escriure per la terminal.

Aixi doncs, anem a crear una llibrería que continga aquestes funcions de calculadora, i que utilitzarem al nostre codi. Cal tindre en compte un aspecte més per tal de definir una llibrería, i és que, **quan creem una llibrería, hem de distingir entre el fitxer de capçalera, amb extensió `.h` i el fitxer d'implementació, amb extensió `.c`**:

- El fitxer de capçalera (`.h`), contindrà les definicions de les funcions, sense cap detall de la seua implementació (diu **què** és el que fan les funcions de llibrería).
- El fitxer d'implementació (`.c`), serà qui realitzarà la implementació de les funcions (diu **com** es fan les funcions de la llibreria). Aquest fitxer **no** implementarà cap funció *main*.
- Després, el fitxer principal, haurà d'incorporar la llibrería amb un *include* del fitxer de capçalera, i ja podrà accedir a les funcions d'aquesta.

Veiem com quedaria l'exemple.

- El fitxer `calc.c` tindrà implementades les funcions:

```c
int suma(int op1, int op2){
    return (op1+op2);
}

int resta(int op1, int op2){
    return (op1-op2);
}

int multiplica(int op1, int op2){
    return (op1*op2);
}

int divideix(int op1, int op2){
    return (op1/op2);
}
```

- El fitxer `calc.h` conté les capçaleres d'aquestes funcions, per tal de ser utilitzades per altre programa:

```c
#ifndef MYCALC
#define MYCALC

int suma(int op1, int op2);
int resta(int op1, int op2);
int multiplica(int op1, int op2);
int divideix(int op1, int op2);

#endif
```

Com veiem, a banda de les funcions, hi ha tres línies que indiquen les *directives del preprocessador* `#ifndef`, `#define` i `#endif`. Aquestes directives actúen com a *guarda* https://en.wikipedia.org/wiki/Include_guard. Amb açò evitem un problema bastant habitual que consisteix en importar una llibrería diverses vegades. Aquestes directives s'encarreguen de no *importar* el codi si aquest ja ha estat importat, de la següent forma:

- `#ifndef MYCALC`: Comprova si s'ha definit prèviament la *macro de guarda* `MYCALC` . Si aquesta comprovació torna cert (i per tant, aquesta macro **no** està definida),segueix el fil d'execució, de forma normal.
- `#define MYCALC`: Estableix la *macro de guarda* `MYCALC` al preprocessador.
- `#endif`: En cas que el test `#ifndef MYCALC` tornara fals (és a dir, que la macro **sí** que està definida), el preprocessador passa directament a aquest alínia, botant-se tot el codi previ.

Amb aquest mecanisme de definir *macros de guarda* a les directives del preprocessador, aconseguim que, si diversos fitxers font inclouen les mateixes llibreríes (com podría ser bé aquesta amb les

funcions de calculadora, o bé una llibreria estàndard com *stdio.h*), aquesta només siga carregada una vegada.

Compilant amb llibreries pròpies

Per tal de compilar i enllaçar l'exemple, hem de tindre en compte la següent relació de dependències entre els fonts:

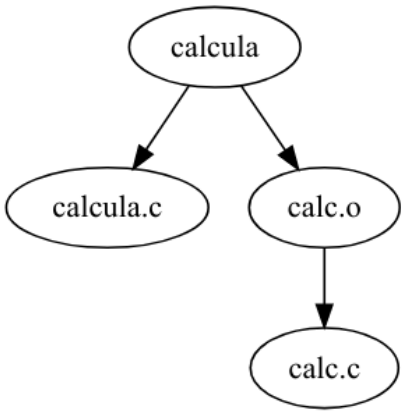

Figura 1: Dependències

És a dir, `calcula` necessita tant del seu codi font `calcula.c` com del fitxer objecte de la llibrería, `calc.o`, i per tal d'obrindre aquest fitxer objecte, necessitem del fitxer font de la llibrería `calc.c`.

Per tal de realitzar la compilació, hauriem de fer:

1. Obtindre el fitxer objecte `calc.o` a partir de `calc.c`:

```
1 gcc -c calc.c -o calc.o
```

Si ens fixem, hem introduït l'opció `-c`. Aquesta opció indica que no execute l'enllaçador o *linker*, i és necessària en el cas de les llibreríes per a que no busque enllaçar la funció *main*. L'eixida que obtindrem serà doncs un fitxer objecte, sense funció principal.

2. Obtenim el fitxer executable `calcula`, a partir de l'objecte `calc.o` i el propi font `calcula.c`:

```
1 gcc calc.o calcula.c -o calcula
```

Com veiem, ara no hem afegit l'opció `-c`, ja que sí que volem que es realitze la fase de *linkat* i obtingam l'executable en sí, enllaçant tots els objecte.

Ara ja podrem executar el nostre programa:

```
$ ./calcula
La suma de 10 i 5 és 15
La resta entre 10 i 5 és 5
La multiplicació de 10 i 5 és 50
La divisió entre 10 i 5 és 2
```

Com veiem, per tal d'obtenir el nostre executable final, hem hagut de compilar, de forma ordenada els diferents fonts.

Quan treballem amb programes que utilitzen múltiples fitxers fonts, aquesta tasca pot complicar-se bastant, i pot ser bastant tediosa. Aci és on entren en joc la potència de l'eina *make*, com vorem a l'apartat següent.

> **TO-DO**
>
> - Creeu i compileu el primer programa de la calculadora per verificar que funciona.
> - Afegiu una nova funció que es diga `major`, i que retornarà quin dels dos números és el major. Imprimirem el resultat com en la resta de casos, de forma que la última línia que mostre siga algo semblant a: `El major entre 10 i 5 és 10`.
> - Una vegada comprovat el funcionament, feu el mateix amb el segon exemple, afegint les modificacions corresponents a cada fitxer.
> - Compileu el segon exemple i verifiqueu que tot funciona correctament.

Creació d'un Makefile

Els *Makefile* són fitxers que s'inclouen en la mateixa carpeta arrel d'un projecte, i li indiquen al programa *make* què ha de fer amb els fitxers font per tal de compilar-los.

El format d'un *Makefile* consisteix en un conjunt de regles, generalment amb la següent forma:

```
target: fitxers_necessaris
    ordre
    ordre
```

On:

- `target`: Indica el nom o noms dels fitxers *objectiu*, és a dir, què es va a crear amb eixa regla. Normalment s'especifica un únic objectiu per regla.
- `fitxers_necessaris`: És la llista de fitxers, separats per espais que necessitem que existisquen per tal de generar el *target*.
- `ordre`: Són les diferents ordres o passos que s'han de realitzar amb els *fitxers_necessaris* per obtenir el *target*. **Aquestes línies comencen amb un tabulador, no amb espais**.

Si tornem ara al gràfic de dependències dels fonts que hem vist abans:

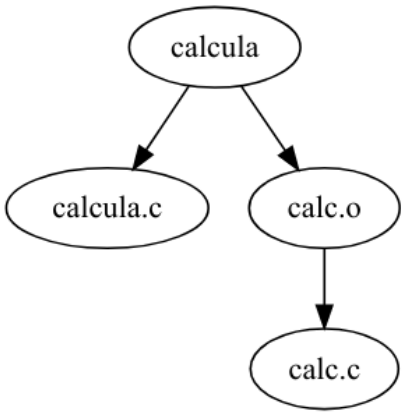

Figura 2: Dependències

Tenim que:

- Per obtenir *calcula* necessitem *calcula.c* i *calc.o*.
- Per obtenir *calc.o* necessitem *calc.c*.

Aleshores, tindrem dues regles, que podrem expressar amb el següent `Makefile`:

```
1 calcula: calcula.c calc.o
2     gcc -Wall -g calcula.c calc.o -o calcula
3
4 calc.o: calc.c calc.h
5     gcc -g -Wall -c calc.c -o calc.o
```

Com veieu, hem afegit a les ordres de compilació els arguments necessaris per tal que ens mostren també els warnings.

Per tal de realitzar la construcció, podem fer ús de `make` de vàries formes:

- `make objectiu`, per construïr un objectiu concret. En aquest cas, podriem fer `make calcula` per generar `calcula`, o `make calc.o` si només volem generar el programa objecte `calc.o`.
- `make`, sense cap objectiu, de manera que per defecte, construïrà el primer objectiu indicat.

Així doncs, en l'exemple, quan fem un `make` o un `make calcula`, estarem construint el *target* `calcula`. Com veiem, aquest necessita el fitxer `calcula.c`, i el `calc.o`. En cas que aquest segon no existisca, **make, automàticament buscarà la regla per generar-lo** i executarà les ordres corresponents. En un principi, no importa l'ordre de les regles, més enllà del requeriment que si no indiquem objectiu, `make` construeix per defecte el primer que indiquem.

> **TO-DO**
>
> - Creeu el fitxer *Makefile* i proveu de construïr els diferents objectius, provant el resultat de `make`, `make calcula` i `make calc.o`. Esborra tots els fitxers objecte i executables entre cada prova.
> - Prova a invertir l'ordre de les regles en el *Makefile*, primer la regla de `calc.o` i després `calcula`. Executa el `make` amb aquest *Makefile* i explica els resultats.
> - Restaura el fitxer *Makefile* amb l'ordre anterior.

Altres tasques per a Make

A mesura que un projecte creix, amb diferents fitxes font, veurem com se'ns generen molts fitxers objecte intermitjos `.o`, que realment, una vegada obtingut l'executable final, no serveixen per a molt.

Per tal de mantenir un projecte *net*, aquests fitxers `.o` haurien de tindre un caràcter temporal. Amb *git* hem vist com podem incloure un fitxer *.gitignore* per a que aquest tipus de fitxers no es puguen als repositoris, però amb *make* podem fer alguna cosa més.

Així doncs, una manera de *netejar* els nostres projectes és esborrar aquests fitxers de forma manual, o també podem crear una tasca dins el *make* que els esborre. Tingueu en compte que les ordres del *Makefile* no han de ser només ordres del `gcc`, sinò que pot ser qualsevol ordre.

Així doncs, aquesta tasca per *netejar* el projecte, que per norma general s'anomena *clean* tindrà el següent aspecte:

```
1  .PHONY: clean
2  clean:
3      rm -rf *.o
```

Fixeu-se que és una miqueta diferent a les tasques que hem vist fins ara, ja que el seu comportament és un poc diferent. En primer lloc, cal recorda que **els noms d'objectius es corresponen amb el nom del fitxer que generen**. Ara bé... l'objectiu `clean` no genera cap fitxer, sinò que els esborra. Per indicar això, cal fer-ho amb la línia `.PHONY: clean`.

Ús de variables al Makefile

Els fitxers makefile admeten l'ús de variables que podem utilitzar al llarg del fitxer. Anem a utilitzar un parell de variables per guardar el nom del compilador `gcc` i els flags `-Wall -g`, de manera que

si en algun moment volem canviar el compilador o afegir flags, no haguem de modificar-ho en tot el fitxer.

Aquestes variables es defineixen al principi del fitxer, i no cal especificar-ne el tipus ni posar-les entre cometes:

```
CC=gcc
CFLAGS=-Wall -g
```

Per tal d'usar-les després al fitxer, caldrà fer-ho amb la sintaxi `$(VARIABLE)`, per exemple:

```
$(CC) $(CFLAGS) fitxers_font -o fitxer_objecte
```

Amb açó, el nostre make quedaria:

```
CC=gcc
CFLAGS=-Wall -g

calcula: calcula.c calc.o
    $(CC) $(CFLAGS) calcula.c calc.o -o calcula

calc.o: calc.c calc.h
    $(CC) $(CFLAGS) -c calc.c -o calc.o

.PHONY: clean
clean:
    rm -rf *.o
    rm calcula
```

Preparant una distribució

Quan treballem en un entorn real, normalment haurem de distribuïr el nostre codi d'alguna manera, per tal de deixar-lo preparat per a la seua instal·lació, o bé per distribuïr el codi, per exemple comprimit.

Per a això, podem utilitzar un codi semblant al següent:

```
.PHONY: dist
dist: clean calcula
    rm -rf ../dist;
    mkdir -p ../dist/usr/bin/calc
    cp calcula ../dist/usr/bin/calc

.PHONY: targz
targz: clean
    mkdir -p source
    cp *.c *.h Makefile source
    tar -cvf calcula.tar source
```

```
12      gzip calcula.tar
13      rm -rf calcula.tar
14      rm -rf source
```

Veiem per parts cadascuna d'aquestes regles:

- **Objectiu `dist`**: Requereix prèviament els objectius *clean* i *calcula*. Les ordres que realitza són:
  - `rm -rf ../dist;`: Esborra, si existeix, la carpeta `/dist` creada prèviament al directori superior.
  - `mkdir -p ../dist/usr/bin`: Crea al directori pare del que estem la ruta `/dist/usr/bin`
  - `cp calcula ../dist/usr/bin/`: Copia l'executable `calcula` dins la carpeta que hem creat.
  - Amb açò, ja tenim la carpeta creada amb els executables.
- **Objectiu `targz`**: Requereix prèviament l'objectiu *clean*. Les ordres que realitza són:
  - `mkdir -p source`: Crea la carpeta source al directori actual.
  - `cp *.c *.h Makefile source`: Copia tots els fitxers `.c`, `.h` i el `Makefile` a la carpeta `source` que hem creat.
  - `tar -cvf calcula.tar source`: Crea el fitxer `calcula.tar` a partir de la carpeta `source`.
  - `gzip calcula.tar`: Comprimeiz en *zip* el fitxer `calcula.tar`.
  - `rm -rf calcula.tar`: Esborra el fitxer intermedi `tar.gz`.
  - `rm -rf source`: Esborra el directori temporal `source`.

A més, podem incloure un altre target per fer la instal·lació al sistema:

```
1  install: dist
2      cp -r ../dist/* /
```

El que faríem amb açò seria indicar que per fer l'install, necessitem executar abans l'objectiu `dist`, i amb aquest fet, copiem tot el contingut del directori `dist` a l'arrel del sistema. És a dir, crearem el fitxer `/usr/bin/calcula`, amb el qual podem utilitzar aquesta ordre des de qualsevol lloc.

> **TO-DO**
>
> - Completeu el vostre makefile amb els targets `dist`, `targz` i `install`, i comproveu el seu funcionament.

Enllaços d'interès

- *Cómo hacer un make*: https://hernandis.me/2017/03/20/como-hacer-un-makefile.html
- *Make tutorial by example*: http://makefiletutorial.com/