MAKEFILE

Para no tener que escribir siempre gcc -Wall -O3...

- Creamos un archivo llamado makefile, sin extensión.

- Este archivo se va a encargar de compilar los programas. Lo único que tenemos que hacer es tenerlo en la misma carpeta que los archivos .c

- Para compilar abrimos la terminal y escribimos ‘make’, bastante más cómodo que escribir cada vez gcc -Wall -O3 -o main.e main.c -lm

Cómo es un makefile:

Usando nuestro editor de texto creamos un archivo llamado makefile, sin extensión, y escribimos por ejemplo lo siguiente.

```
all: helloworld.e

helloworld.e: helloworld.c
    gcc -Wall -O3 -o helloworld.e helloworld.c -lm

clean:
    rm helloworld.e
```

Estructura del makefile:

target: prerequisitos
    Cómo armar el target con los prerequisitos

Lo probamos para compilar el programa helloworld.c

```
francomayo@MacBook-Air-de-Franco /V/S/F/F/F/A/C/M/makefile_simple (master)> ls
helloworld.c  makefile
francomayo@MacBook-Air-de-Franco /V/S/F/F/F/A/C/M/makefile_simple (master)> make
gcc -Wall -O3 -o helloworld.e helloworld.c -lm
francomayo@MacBook-Air-de-Franco /V/S/F/F/F/A/C/M/makefile_simple (master)> ls
helloworld.c  helloworld.e* makefile
francomayo@MacBook-Air-de-Franco /V/S/F/F/F/A/C/M/makefile_simple (master)> make clean
rm helloworld.e
francomayo@MacBook-Air-de-Franco /V/S/F/F/F/A/C/M/makefile_simple (master)> ls
helloworld.c  makefile
francomayo@MacBook-Air-de-Franco /V/S/F/F/F/A/C/M/makefile_simple (master)>
```

Vemos que efectivamente el comando make nos crea el ejecutable, y el comando make clean lo elimina.

Para un caso más general donde quiero compilar un único archivo:

```
all: main.e

SOURCE_C = $(wildcard *.c) #La función
#wildcard busca todos los archivos .c
#que haya en la carpeta y los guarda en
#la variable SOURCE_C
main.e: $(SOURCE_C)
    gcc -Wall -O3 -o $@ $^ -lm

clean:
    rm main.e
```

En este caso makefile busca el nombre de .c que haya en la carpeta y genera el ejecutable main.e, no hace falta aclarar el nombre del archivo.

@ indica el target y ^ los prerequisitos dentro de cada regla.

Compilar muchos archivos en un único ejecutable:

```
1  .PHONY: default objects executable all clean
2  CC = gcc
3  CC_FLAGS = -Wall -O3
4  LD_FLAGS = -lm
5  LD = $(CC)
6  SOURCE_C = $(wildcard *.c)
7  OBJECTS_C = $(patsubst %.c, %.o, $(SOURCE_C))
8  EXECUTABLE = main.e
9  default: all
10 objects: $(OBJECTS_C)
11 executable: $(EXECUTABLE)
12 all: objects executable
13 %.o: %.c
14   $(CC) $(CC_FLAGS) -c $^ -o $@
15 $(EXECUTABLE): $(OBJECTS_C)
16   $(LD) -o $@ $^ $(LD_FLAGS)
17 clean:
18   rm $(OBJECTS_C) $(EXECUTABLE)
```

Creo variables por comodidad

.PHONY indica todas las cosas que puede hacer el makefile. Por default hace sólo la primera. Para usar las otras tengo que usar el comando make objects (por ejemplo)

Esta lista genera los objetos .o para todos los .c

Defino los targets

Estas líneas se encaran de compilar el programa, primero crean los objetos y luego los combinan en el ejecutable

En resumen:

Teniendo un makefile podems compilar los programas, desde los más simples a los más complicados simplemente escribiendo ‘make’ en la terminal.

Esto está muy bueno para compilar códigos hechos en módulos, ya que con un sólo comando compilamos todo.

Es una forma más prolija de hacer las cosas. Sobre todo si quiero compartir mi código. Queda todo más “empaquetado”.

Con tener un el último makefile que vimos hecho y copiarlo a las carpetas donde sea necesario ya les alcanza para compilar cualquier código modular que tengan (Salvo que necesiten flags extra, por ejemplo si usan OpenMP o cosas parecidas)

Comentarios:

Para compilar código de varios archivos, cada archivo debe tener los #include necesarios, sino va a aparecer un error cuando el compilador quiera hacer los objetos.

No hay que incluir más los .c al final del archivo del main.

De este modo si modifican algún arcihvo, sólamente vuelve a compilar ese archivo, los otros ya quedan. Makefile hace esto automáticamente, es más prolijo para no andar compilando cosas que no hagan falta.