









1 Repaso

Repaso: ¿Cómo compilamos nuestros programas en C?

```
1 gcc -Wall -Wextra -Werror -pedantic -std=c99 -g -c main.c array_helpers.c array.c
2 gcc -Wall -Wextra -Werror -pedantic -std=c99 -g -o parser main.o array_helpers.o
   array.o
```

¿Qué problemas tiene esto?

- Nos podemos olvidar algún flag por error
- Podemos sobrescribir un archivo que no queremos
- Estamos haciendo         a cada rato
- Podemos llegar a correrlos en el orden incorrecto
- **No escala**

¿Qué soluciones hay?

2 make y Makefiles

¿Qué es make? ¿Y los Makefiles?


Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.¹

- **make** es una utilidad presente en los sistemas Linux (y tipo Unix)
- Nos permite automatizar la compilación de nuestro programa
- Para usarla debemos:
 1. Crear un **Makefile**
 2. Ejecutar desde la terminal **make**
- El **Makefile** es el archivo de configuración que le dice cómo compilar nuestro código
- Usar **make** (o herramientas similares) evita errores y ahorra tiempo

Cómo escribir un Makefile

- Debemos ubicarnos (con **cd**) en el directorio de nuestro proyecto
- Creamos un archivo de nombre **Makefile**
- Este archivo se escribe de la siguiente manera:

```
1 VARIABLE = a b c
2 # OBJETIVO : DEPENDENCIAS
3 ejecutable1: archivo1 ar2 ar3
4     comando1 $(VARIABLE)
5     comando2 $^ #comando2 archivo1 ar2 ar3
6
7 ar2: ar4
8     comando3 $@ #comando3 ar2
9 #...
```

¹Proyecto GNU Make  (<http://savannah.gnu.org/projects/make/>)

- Luego podremos ejecutar `make` como:

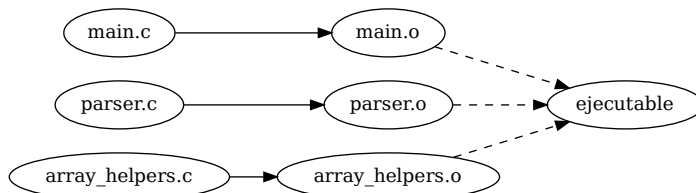
```

- $ make <objetivo>
- $ make

```

¿Y para qué sirve esto?

- Nos permite definir qué cosas dependen de qué otras cosas (por ejemplo: el ejecutable depende de los archivos objeto, y éstos de los `.c`)
- En base a esas dependencias `make` hace sólo lo necesario
 - Si editamos un `.c` sólo recompila ese archivo, no todos
 - Si no modificamos nada desde la última vez nos avisa eso y termina
- Si falla alguna dependencia nos avisa del error y se detiene



3 Ejemplos

Construyamos uno (Pt. 1)

¿Cómo escribimos un Makefile para el programa del gráfico de recién?

```

1 parser: main.c array_helpers.c parser.c
2     gcc -Wall -Wextra -Werror -pedantic -std=c99 -g -c main.c array_helpers.c
3     gcc -Wall -Wextra -Werror -pedantic -std=c99 -g -o parser main.o
4     array_helpers.o parser.o

```

¿Y si usamos variables?

```

1 CC = gcc
2 CFLAGS = -Wall -Wextra -Werror -pedantic -std=c99 -g
3
4 parser: main.c array_helpers.c parser.c
5     $(CC) $(CFLAGS) -c main.c array_helpers.c parser.c
6     $(CC) $(CFLAGS) -o parser main.o array_helpers.o parser.o

```

Construyamos uno (Pt. 2)

¿Y si modularizamos?

```

1 CC = gcc
2 CFLAGS = -Wall -Wextra -Werror -pedantic -std=c99 -g
3 parser: main.o array_helpers.o parser.o
4     $(CC) $(CFLAGS) -o parser main.o array_helpers.o parser.o
5 main.o: main.c
6     $(CC) $(CFLAGS) -o main.o -c main.c
7 array_helpers.o: array_helpers.c

```

```

8      $(CC) $(CFLAGS) -o array_helpers.o -c array_helpers.c
9  parser.o: parser.c
10     $(CC) $(CFLAGS) -o parser.o -c parser.c

```

¿Podemos quitar esta repetición?

```

1  CC = gcc
2  CFLAGS = -Wall -Wextra -Werror -pedantic -std=c99 -g
3  parser: main.o array_helpers.o parser.o
4      $(CC) $(CFLAGS) -o $@ $^
5  %.o: %.c
6      $(CC) -c -o $@ $< $(CFLAGS)

```

Makefile definitivo

Un buen punto de partida para un proyecto básico de C

```

1  CC = gcc
2  CFLAGS = -Wall -Wextra -Werror -pedantic -std=c99 -g
3  #LDFLAGS = -lm
4  HEADERS = $(wildcard *.h)
5  SOURCES = $(wildcard *.c)
6  OBJECTS = $(SOURCES:.c=.o)
7  TARGET = mi_ejecutable
8
9  $(TARGET): $(OBJECTS)
10     $(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)
11
12 %.o: %.c $(HEADERS)
13     $(CC) -c -o $@ $< $(CFLAGS)
14
15 .PHONY: clean
16
17 clean:
18     rm -f $(OBJECTS) $(TARGET)

```

4 Extras

Más recursos

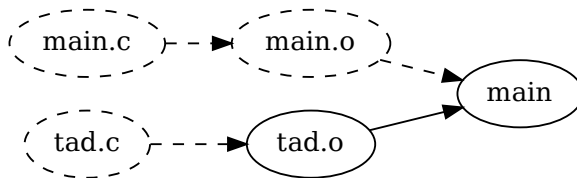
- Makefiles del mundo real:
 - **XV6** <https://github.com/mit-pdos/xv6-public/blob/master/Makefile>
 - **Linux 0.01** <https://github.com/zaug/linux-0.01/blob/master/Makefile>
 - **Linux** <https://github.com/torvalds/linux/blob/master/Makefile>
 - **FreeBSD** <https://cgit.freebsd.org/src/tree/Makefile>
 - **Vim** <https://github.com/vim/vim/blob/master/Makefile>
 - **Emacs** <https://github.com/emacs-mirror/emacs/blob/master/GNUMakefile>
 - **Python** <https://github.com/python/cpython/blob/main/Makefile.pre.in>
- Un tutorial súper-completo <https://makefiletutorial.com/>
- Un tutorial bien breve, con ejemplos para copiar <https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>
- Wikipedia en español <https://es.wikipedia.org/wiki/Make>

Makefile mínimo

Aprovechamos las *reglas implícitas* de `make`

```
1 CC = gcc
2 CFLAGS = -Wall -Wextra -Werror -pedantic -std=c99 -g
3
4 main: tad.o
5
6 .PHONY: clean
7 clean:
8     rm -f *.o *~ main
```

- Si tenemos un *objetivo* que es el nombre de un ejecutable (sin extensión), automáticamente *dependerá* del `.o` del mismo nombre, e intentará generarlo linkeando sus dependencias
- Si tenemos un *objetivo* que es un `.o` *dependerá* de su `.c` y se compilará a partir del mismo
- En ambos casos esto se hace usando las variables `CC` y `CFLAGS`



Meta: un Makefile para una presentación

Dato de color: Esta presentación fue generada con el siguiente Makefile

```
1 all: slides.pdf handout.pdf
2
3 %.pdf: %.tex main.tex Makefile-*
4     mkdir -p .out
5     cd .out; TEXINPUTS="...:" pdflatex -shell-escape ../$<
6     ln -f .out/$@ $@
7
8 .PHONY: all clean show
9
10 clean:
11     rm .out/*
12     rmdir .out
13
14 show: all
15     xdg-open slides.pdf
16     xdg-open handout.pdf
```