Make

Automatización del Proceso de Traducción

Esp. Ing. José María Sola, profesor

Revisión 2.1.0
2025-06-06

Tabla de contenidos

1

Introducción a *make*

Un proyecto informático con aplicación práctica en la industria, que sea que más un mínimo proyecto ejemplificativo, está formado por *muchos* archivos fuente. La compilación de estos proyectos se requiere escribir varios comandos y gestionar las dependencias entre los fuentes. El volumen de archivos en el proyecto es entonces un problema a tratar. Para traer un poco de perspectiva, la *calculadora de Windows* está formada por 500 archivos fuente, *Chrome* por 20 mil, *Linux* por 100 mil, *gcc* por 120 mil, *clang* por 150 mil, *Chromium* por 200 mil, *WebKit* por 330 mil, y el premio al repositorio más grande del mundo se lo lleva *Windows* con 3,5 millones de archivos fuente.

Claramente, la gestión manual no es eficiente para esta escala. La utilidad `make` junto con los *makefiles* proponen una solución. En este texto vamos a ver para qué es y como se usa la utilidad `make`.

1.1. ¿Qué es *make*?

Es una herramienta que determina automáticamete que partes de un programa o sistema grande formado por varios componentes necesitan recomplilarse, y emite los comandos para hacerlo. Efectivamente *hace* o *fabrica* (i.e., *makes*) el programa.

1.2. ¿Qué facilita *make*?

La actualización automática de archivos desde otros archivos, que se disparan cuando los segundos se modifican. Automatiza el proceso de *building* (i.e., "*buildeo*" o traducción) de un programa o sistema grande, formado por varios

archivos. Permite actualizar solo lo que cambió, sin necesidad de recompilar todos los archivos fuente que componen el programa.

Ejercicio 1. Busque proyectos reconocidos que utilicen *Makefiles*. Por ejemplo, ¿dónde está el *Makefile* para *Linux*?

Ejercicio 2. Investigue sobre alternativas a `make`.

1.3. ¿Qué es un *makefile*?

Un `makefile` es un archivo con una notación declarativa que especifica las dependencias, relaciones, y comandos para construir y actualizar uno o más productos. Si la especificación se encuentra en la carpeta actual, la simple invocación a `make` construye los productos.

En un programa, típicamente, un ejecutable es actualizado desde archivos objetos, que su vez son *hechos* (i.e., *made*) mediante la compilación de archivos fuentes.

El programa `make` utiliza las especificacione en el *makefile* y las fechas de modificación de los archivos objetivo y archivos prerequeisitos para decidir qué archivos necesitan actualizarse, es decir, volver a *fabricarlos*. Para cada uno de esos archivos, emite las recetas definidas en el *makefile*.

1.4. ¿Qué sintaxis tiene un *makefile*? ¿Qué partes tiene una regla?

Por convención, al *makefile* de un proyecto se lo nombra `Makefile`, con la `M` Del punto de vista más fundamental, un *makefile*, es una secuencia de reglas. mayúscula para aprovechar el ordenamiento de los archivos y sin extensión. Las reglas son la estructura principal, y esta es su sintaxis:

```
objetivo ... : prerequisitos ...
    comandos
    ...
    ...
```

Los comandos deben estar precedidos por exactamente un caracter tabulado, y no por espacios.

Un mismo archivo *makefile* puede tener múltiples reglas.

> **Ejercicio 3**. Especifique formalmente, pero de manera simple la sintaxis de una regla, no es necesario refleje todas las alternativas válidas de reglas.

1.5. ¿Qué significan las regla de *make*? ¿Cuál es su semántica?

Las reglas tienen el siguiente significado:

- que el resultado *objetivo* depende de los *prerequisitos*,
- que el objetivo se produce siguiendo la receta formada por *comandos*.
- que si los *prerequisitos* están más actualizados que los *objetivos* que porducen, se vuelven a generar los *objetivos*.

Las reglas se evalúan ejecutando el programa `make`, y el resultado va a ser la *fabricación* (i.e., `make`) de los objetivos.

La utilidad `make` lee las dependencias declaradas en el *makefile* y determina que componentes de la solución fueron actualizados desde la última vez que

se construyó el producto, `make` reconstruye solo los componentes que fueron actualizados y reconstruye el producto.

Ejercicio 4. Suponga el siguiente *Makefile*:

```
A.o: A.c A.h
    cc -c A.c -o A.o
```

1. ¿Qué ocurre si corremos `make` con ese *Makefile*?
2. ¿Qué ocurre si lo corremos de nuevo?
3. ¿Qué ocurre si hacemos una modificación en la interfaz (`A.h`) y ejecutamos `make` de nuevo?

1.6. ¿Como puedo correr *make*?

Por defecto, `make` busca la especificación de construcción un archivo llamado `Makefile`. Si se necesita llamarlo de otra manera o se necesita tener más de una especificación, `make` acepta la opción `-f`. Para simplificar el proceso la buena práctica es contener los archivos fuente y el *makefile* en una misma carpeta. Estas son algunas de las maneras que se puede ejecutar el programa `make`:

1. Archivo *makefile* implícito:
   `make`
   En esta versión `make` busca archivos *makefile* con nombres estándard, la convención más común es llamar al archivo `Makefile`. +En sistemas *Windows* es puede ser necesario usar el comando `mingw32-make` para ejecutar `make`.
2. Archivo *makefile* explícito:
   `make -f` *filename*
   La opción `-f` le indica a `make` que tome las reglas del archivo *filename*.
3. Generación de un objetivo en particular:
   `make` *objetivos*
   `make -f` *filename objetivos*
   Se fabrican los objetivos indicados.

Ejemplos de corridas `make`:

- `make`
- `make hello`
- `make app`
- `make -f reglas app`
- `make appios`
- `make appandroid`
- `make appois appandroid`
- `make run`
- `make test`
- `make install`
- `make clean`

2

Casos de Estudio

2.1. Hagamos una Hamburguesa

Este es un *makefile* simple para hacer hamburguesas. Se corre simplemente con el comando `make`.

```
hamburguesa: ingredientes
 @echo "Cocinar patty"
 @echo "Cocinar panceta"
 @echo "Calentar panes"
 @echo "Armar hamburguesa"

ingredientes:
 @echo "Conseguir pan, patty, y panceta"
```

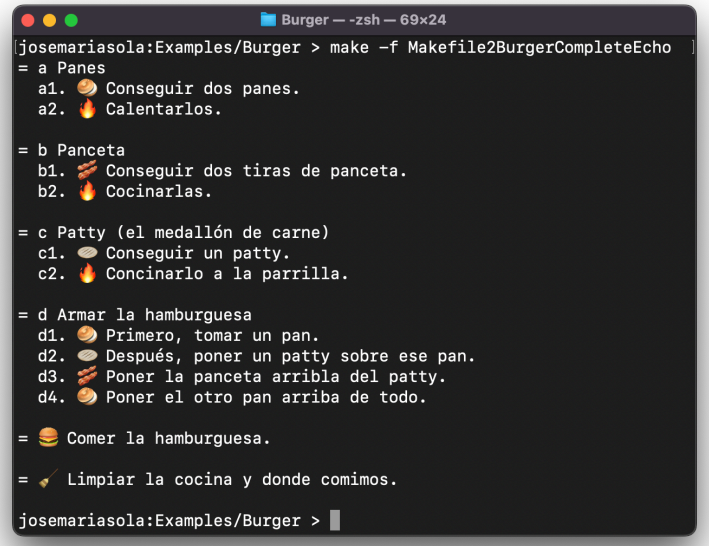
Esta segunda versión versión, que es un poco más detallada, hace uso de:

- Comentarios.
- Múltiples objetivos.
- Objetivos *falsos*.

```
.PHONY: all hamburguesa run clean

all: hamburguesa run clean

# El producto principal
hamburguesa: panes panceta patty
 @echo "= d Armar la hamburguesa"
 @echo "  d1. # Primero, tomar un pan."
```

```
 @echo "  d2. # Después, poner un patty sobre ese pan."
 @echo "  d3. # Poner la panceta arribla del patty."
 @echo "  d4. # Poner el otro pan arriba de todo."
 @echo ""

panes:
 @echo "= a Panes"
 @echo "  a1. # Conseguir dos panes."
 @echo "  a2. # Calentarlos."
 @echo ""

panceta:
 @echo "= b Panceta"
 @echo "  b1. # Conseguir dos tiras de panceta."
 @echo "  b2. # Cocinarlas."
 @echo ""

patty:
 @echo "= c Patty (el medallón de carne)"
 @echo "  c1. # Conseguir un patty."
 @echo "  c2. # Concinarlo a la parrilla."
 @echo ""

# La ejecución del producto, por convención se llama run.
run: hamburguesa
 @echo "= # Comer la hamburguesa."
 @echo ""

# La limpeiza posterior, por convesión se llama clean.
clean:
 @echo "= # Limpiar la cocina y donde comimos."
 @echo ""
```

Esta es la salida que se en la terminal al ejecutar `make`:



Ejercicio 5. Investigue qué son los objetivos *falsos* y cuáles son los que por convención son más comunes.

Ejercicio 6. Modifique el *makefile* para que la hamburguesa tenga queso.

Ejercicio 7. Modifique el *makefile* para que sea una opción que tenga tenga queso y que la cantidad de tiras de panceta se pueda especificar; para eso investigue variables.

2.2. *Hello* de Nuevo

Vamos a usar el clásico ejemplo de *Hello, World!* para ver de que formas se puede generar el ejecutable usando `make`.

```
#include <stdio.h>

int main(){
  printf("Hello, World!\n");
}
```

En todas la versiones las reglas están en el archivo llamado `Makefile` y `hello.c` está en la misma carpeta.

2.2.1. Simple

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

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

En esta versión el ejecutable depende del objeto y se dan comandos para crear cada uno. El ejecutable se genera con el simple comando `make`.

2.2.2. Phonies

```
.PHONY: run clean

run: hello
 ./hello

clean:
 rm hello hello.o

hello: hello.o
 cc hello.o -o hello

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

Agregamos un objetivo falso para poder ejecutar el producto. Se ejecuta con simple comando `make run`. Si el ejecutable `hello` está desactulizado lo vuelve a fabricar. También agregamos el objetivo falso `clean` para borrar los productos y subproductos generados.

2.2.3. Recetas por Defecto

```
.PHONY: run clean

run: hello
 ./hello

clean:
 rm hello hello.o

hello: hello.o

hello.o: hello.c
```

`Make` ya conoce las recetas para ejecutables y objetos, las podemos obviar.

2.2.4. Reglas por Defecto

```
.PHONY: run clean

run: hello
 ./hello

clean:
 rm hello hello.o
```

`Make`ya conoce las reglas que indican que los ejecutables tienen como prerequisito a los objetos, y los objetos a los fuentes. También podemos obviarlas.

2.2.5. Sin Makefile

Como `make` sabe como generar ejecutables a partir de fuentes, pasando por objetos, es posible no tener un *makefile* y simplemente usar el comando `make hello`. Esto solo funciona para programa muy simples, casi triviales, no para sistemas con varios módulos y dependencias. Esa situación se trata en el siguiente caso.

2.3. Conversión de Temperaturas

Uno de los usos más comunes de `make` es para la construcción de programas con módulos que exponen interfaces, con una clara separación entre *contratos*, *consumidores*, y *proveedores*. Este caso de estudio ejemplifica la situación con el programa de conversión de tempraruras presentado en el texto ???. El programa está formado por tres archivos:

1. **Contrato**
   El archivo `Conversion.h` establece como usar la función de conversión.
2. **Consumidor**
   El archivo `FahrCel.c` consume la función.
3. **Proveedor**
   El archivo `Conversión.c` provee la función.

```
#ifndef CONVERSION_H_INCLUDED
#define CONVERSION_H_INCLUDED

double Celsius(double);

#endif
```

```
#include <stdio.h>
#include "Conversion.h"

int main(){
 constexpr int LOWER = 0,   // lower limit of table
               UPPER = 300, // upper limit
               STEP  = 20;  // step size

 for(int fahr = LOWER; fahr <= UPPER; fahr += STEP)
  printf("%3d %6.1f\n", fahr, Celsius(fahr) );
}
```

```
#include "Conversion.h"

double Celsius(double f){
 return (5.0/9.0)*(f-32);
}
```

`Make` toma la especificación de las dependencias y de los productos y subproductos a generar de un archivo *Makefile*.

```
FahrCel : FahrCel.o Conversion.o
 cc FahrCel.o Conversion.o -o FahrCel

FahrCel.o : FahrCel.c Conversion.h
 cc -std=c23 -c FahrCel.c -o FahrCel.o

Conversion.o: Conversion.h Conversion.c
 cc -std=c23 -c Conversion.c -o Conversion.o

.PHONY : run clean

run : FahrCel
 ./FahrCel

clean :
 rm -f FahrCel.o Conversion.o FahrCel
```

En la especificación se explicíta que el consumidor depende del proveedor, y que ambos dependen del contrato.

Para ejecutar la especificación, es necesario crear el archivo `Makefile` con el anterior contenido, y ubicarlo en la misma carpeta que los tres archivos fuente. Para constuir el ejecutable del cliente, es sufiente con escribir el comamndo `make` en la línea de comando.

Esta segunda versión del *makefile* utiliza variables tipo *macro* para ser menos repetitivo:

```
BIN      = FahrCel
OBJ      = FahrCel.o Conversion.o
CFLAGS   = -std=c23 -Weverything
RM       = rm -f

$(BIN) : $(OBJ)
 $(CC) $(OBJ) -o $(BIN) $(CFLAGS)

FahrCel.o : FahrCel.c Conversion.h
 $(CC) -c FahrCel.c -o FahrCel.o $(CFLAGS)
```

```
Conversion.o: Conversion.c Conversion.h
 $(CC) -c Conversion.c -o Conversion.o $(CFLAGS)

.PHONY : run clean

run : $(BIN)
 ./$(BIN)

clean :
 $(RM) $(OBJ) $(BIN)
```

Ejercicio 8. Investigue la opción1 `-MM` de *gcc* o *clang* para automatizar la detección de dependencias.

Ejercicio 9. Escriba un *Makefile* para un ejecutable que depende de un fuente y dos módulos. Dé nombres significativos a los archivos, de tal forma que representen la solución a un posible problema real.

3

Referencia y Lectura Adicional

La utilidad `make_ tiene decenas de funcionalidades, esta es solo alguna de las referencias para profundizarlas.

[Interfaces] José María Sola. *Interfaces: Los Contratos entre Proveedores y Consumidores* (2016) https://josemariasola.wordpress.com/ssl/papers#Interfaces

[Make_Mrbook] Hector Urtubia (Mrbook's stuff) *Makefiles: A Tutorial by Example* (2008) https://web.archive.org/web/20201104213646/http://mrbook.org/blog/tutorials/make/

[Make_Maxwell] Bruce A. Maxwell *A Simple Makefile Tutorial* (2016) https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

[GNUMake] Richard M. Stallman, Roland McGrath, Paul D. Smith *GNU Make: A Program for Directing Compilation* (1988) https://www.gnu.org/software/make/manual/make.pdf

[Make_Crawford] Michael Crawford *GenericMakefile* (2014) https://github.com/mbcrawfo/GenericMakefile

[Make_Jimenez] Ricardo Catalinas Jiménez *MagicMakefile* (2011) https://github.com/jimenezrick/magic-makefile

[Make_Penny] David A. Penny *How to use makefiles for automated testing* (2005) http://www.cs.toronto.edu/~penny/teaching/csc444-05f/maketutorial.html

Changelog

2.1.0+2025-06-06

- Added: Missing example, brought from ex *Interfaces & Make*.
- Added: Bibliography.

2.0.0+2025-06-05

- Major overhaul.
- Fixed: Typos.
- Fixed: `-c`
- Added: Burger.
- Added: Hello.
- Added: Temperature.

1.0.0+2022-12-04

- Versión inicial.