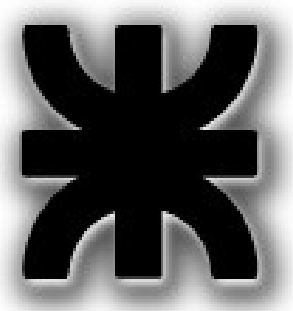


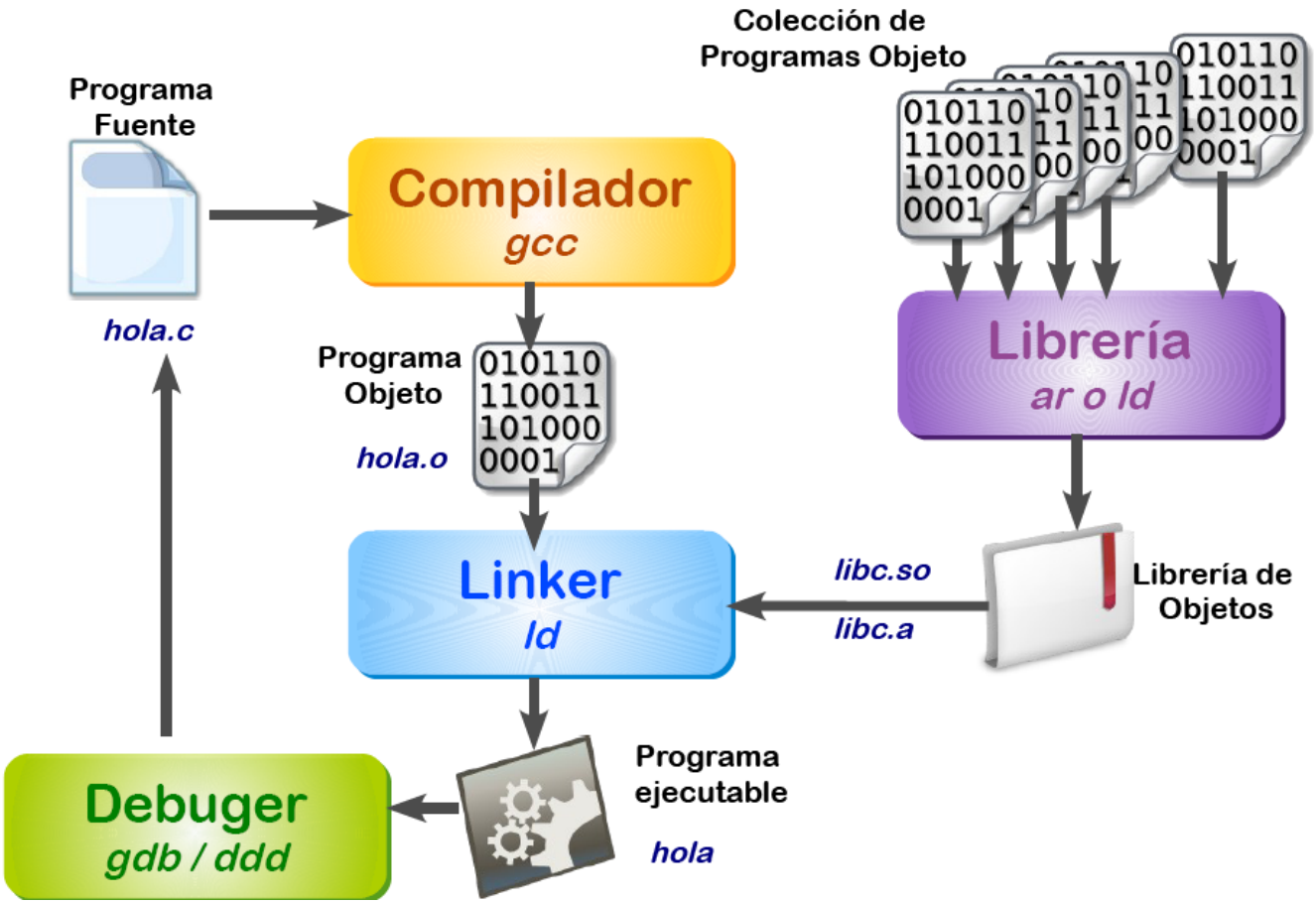
Make - Automatizando el desarrollo

Alejandro Furfaro

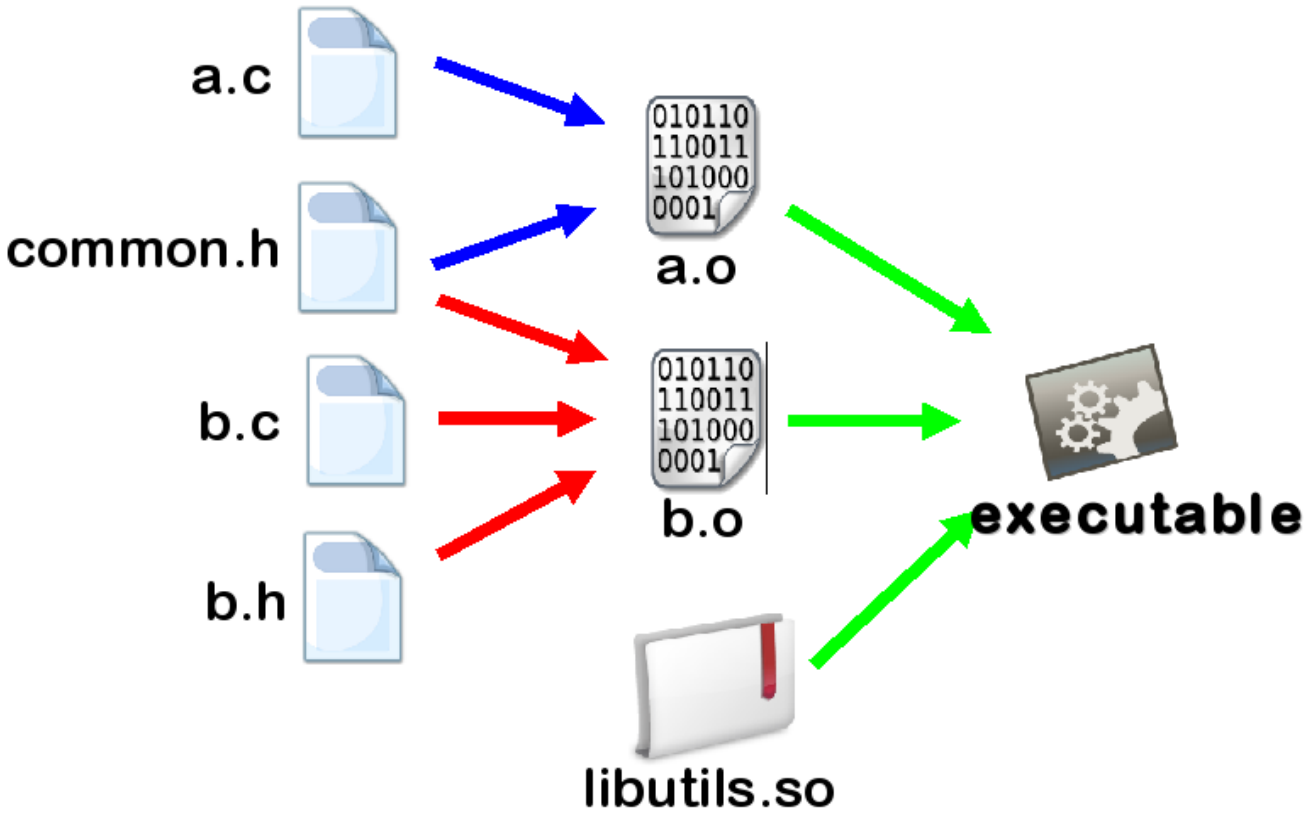
Marzo 2012

Temario

Proceso de desarrollo



Árbol de Dependencias



Propósito

- Make es una herramienta que permite ejecutar una secuencia de procesos.
- Utiliza un script, llamada comúnmente ***Makefile***.
- Es capaz de determinar automáticamente cuales pasos de una secuencia deben repetirse debido al cambio en algunos de los archivos involucrados en la construcción de un objeto, o en una operación.
- Usos mas comunes
  - Recompilar programas que residen en diversos archivos.
  - Testing de programas.

Primeros conceptos

- De manera sencilla lo que tenemos que hacer es definir un archivo llamado ***Makefile*** en el directorio raíz de nuestro proyecto (en realidad lo podemos poner en otro lado) y dentro de ese archivo escribimos las reglas necesarias para construir nuestro proyecto.
- Luego, alcanza con ejecutar

```
make <regla>
```

en el directorio en el que definimos el ***Makefile***.

Ejemplo

```
ejecutable: a.o b.o
    gcc -g a.o b.o -o ejecutable
a.o: a.c
    gcc -g -O0 -c a.c -o a.o
b.o: b.c
    gcc -g -O0 -c b.c -o b.o
clean:
    rm -f ./*.o
    rm -f ejecutable
new: clean ejecutable
```

¡Importante!

Cada una de las sentencias que componen una regla comienzan con el caracter Tabulador. De otro modo **¡No Funciona!**

Ejemplo

```
ejecutable: a.o b.o
    gcc -g a.o b.o -o ejecutable
a.o: a.c
    gcc -g -O0 -c a.c -o a.o
b.o: b.c
    gcc -g -O0 -c b.c -o b.o
clean:
    rm -f ./*.o
    rm -f ejecutable
new: clean ejecutable
```

¡Importante!

Cada una de las sentencias que componen una regla comienzan con el caracter Tabulador. De otro modo **¡No Funciona!**

El mismo ejemplo, pero utilizando variables

```
CC=gcc
ejecutable: a.o b.o
	$(CC) -g a.o b.o -o ejecutable
a.o: a.c
	$(CC) -g -O0 -c a.c -o a.o
b.o: b.c
	$(CC) -g -O0 -c b.c -o b.o
clean:
	rm -f ./*.o
	rm -f ejecutable
new: clean ejecutable
```

CC

La llamamos de este modo por C Compiler. De este modo si quisiéramos cambiar el compilador, tan solo necesitamos retocar el valor de CC.

El mismo ejemplo, pero utilizando variables

```
CC=gcc
ejecutable: a.o b.o
	$(CC) -g a.o b.o -o ejecutable
a.o: a.c
	$(CC) -g -O0 -c a.c -o a.o
b.o: b.c
	$(CC) -g -O0 -c b.c -o b.o
clean:
	rm -f ./*.o
	rm -f ejecutable
new: clean ejecutable
```

CC

La llamamos de este modo por C Compiler. De este modo si quisiéramos cambiar el compilador, tan solo necesitamos retocar el valor de CC.

El mismo ejemplo, con mas variables

```
CC=gcc
CFLAGS=-g -O0 -c
LDFLAGS=-g
OBJS=a.o b.o
ejecutable: $(OBJS)
	$(CC) $(LDFLAGS) $(OBJS) -o ejecutable
a.o: a.c
	$(CC) $(CFLAGS) a.c -o a.o
b.o: b.c
	$(CC) $(CFLAGS) b.c -o b.o
clean:
	rm -f ./*.o
	rm -f ejecutable
new: clean ejecutable
```

Macros

- **$@** Nombre completo del target.
- **$?** lista de las dependencias que se encuentran desactualizadas.
- **$<** El archivo fuente de la dependencia actual.

El mismo ejemplo con variables y macros

```
CC=gcc
CFLAGS=-g -O0 -c
LDFLAGS=-g
OBJS=a.o b.o
ejecutable: $(OBJS)
	$(CC) $(LDFLAGS) $(OBJS) -o $@
a.o: a.c
	$(CC) $(CFLAGS) $< -o $@
b.o: b.c
	$(CC) $(CFLAGS) $< -o $@
clean:
	rm -f ./*.o
	rm -f ejecutable
new: clean ejecutable
```

El ejemplo mas críptico... ¡pero mas versatil!

```
CC=gcc
CFLAGS=-g -O0 -c
LDFLAGS=-g
OBJS=a.o b.o
ejecutable: $(OBJS)
	$(CC) $(LDFLAGS) $(OBJS) -o $@
%.o: %.c
	$(CC) $(CFLAGS) $< -o $@
clean:
	rm -f ./*.o
	rm -f ejecutable
new: clean ejecutable
```

¿Que hace el caracter %?

Significa "cualquier target". Es decir, cualquier archivo que se requiera con terminación **".o"** se crea a partir de su homónimo finalizado en **".c"**.

El ejemplo mas críptico... ¡pero mas versatil!

```
CC=gcc
CFLAGS=-g -O0 -c
LDFLAGS=-g
OBJS=a.o b.o
ejecutable: $(OBJS)
	$(CC) $(LDFLAGS) $(OBJS) -o $@
%.o: %.c
	$(CC) $(CFLAGS) $< -o $@
clean:
	rm -f ./*.o
	rm -f ejecutable
new: clean ejecutable
```

¿Que hace el caracter %?

Significa “cualquier target”. Es decir, cualquier archivo que se requiera con terminación **“.o”** se crea a partir de su homónimo finalizado en **“.c”**.

Para que recompile todo si cambió el propio Makefile...

```
CC=gcc
CFLAGS=-g -O0 -c
LDFLAGS=-g
OBJS=a.o b.o
ejecutable: $(OBJS)
	$(CC) $(LDFLAGS) $(OBJS) -o $@
%.o: %.c Makefile
	$(CC) $(CFLAGS) $< -o $@
clean:
	rm -f ./*.o
	rm -f ejecutable
new: clean ejecutable
```

Agregando comandos para empaquetar y distribuir

```
CC=gcc
CFLAGS=-g -O0 -c
LDFLAGS=-g
OBJS=a.o b.o
SOURCES=a.c b.c
HEADERS=*.h
ejecutable: $(OBJS)
	$(CC) $(LDFLAGS)  $(OBJS) -o $@
%.o: %.c Makefile
	$(CC) $(CFLAGS) -c $< -o $@
clean:
	rm -f ./*.o
	rm -f ejecutable
new: clean ejecutable
entrega: $(SOURCES) $(HEADERS) Makefile
	tar zcvf entrega.tar.gz $(SOURCES) $(HEADERS) Makefi
```

Super genérico

```
CC=gcc
CFLAGS=-g -O0 -c
LDFLAGS=-g
OBJS=a.o b.o
SOURCES=$(OBJS:.o=.c)
HEADERS=*.h
ejecutable: $(OBJS)
	$(CC) $(LDFLAGS)  $(OBJS) -o $@
%.o: %.c Makefile
	$(CC) $(CFLAGS) -c $< -o $@
clean:
	rm -f ./*.o
	rm -f ejecutable
new: clean ejecutable
entrega: $(SOURCES) $(HEADERS) Makefile
	tar zcvf entrega.tar.gz $(SOURCES) $(HEADERS) Makefi
```

¿Que aprendimos?

1. A automatizar el proceso de desarrollo
2. A usar los rudimentos de la sintaxis de make para desarrollar todo tipo de programas
3. A parametrizar los scripts (tarea escalable a los programas)