TUTORIAL 5
USO DE MAKE Y ESCRITURA DE MAKEFILES

Prof. Ing. Miguel Angel Aguilar Ulloa

TEC

LICENCIA

Usted es libre de:

✓ Copiar, distribuir y comunicar públicamente la obra.

✓ Hacer obras derivadas.

Bajo las siguientes condiciones:

✓ Reconocimiento — Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).

✓ No comercial — No puede utilizar esta obra para fines comerciales.

✓Compartir bajo la misma licencia — Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

Texto de la licencia: http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode.

Contenido

1. GNU Make.
2. Proceso de compilación: uno o varios archivos.
3. Makefile
4. Optimizaciones de los Makefile’s.
5. Ejemplos de Makefile’s

1. GNU MAKE

GNU Make - http://www.gnu.org/software/make

• Es una herramienta de generación o automatización de código, muy usada en los sistemas operativos tipo Unix/Linux. La utilidad **`make`** determina automáticamente que piezas de código de un programa necesitan ser compiladas o recopiladas y ejecuta comandos para ello.

• Por defecto lee las instrucciones para generar el programa u otra acción del archivo llamado **`makefile`**. Las instrucciones escritas en este archivo se llaman reglas.

• La mayoría de proyectos open source utilizan a **`make`** como la herramienta para la construcción de los mismo.

• El manual completo de makefile lo puede encontrar en: http://www.gnu.org/software/make/manual/make.pdf

Funciones de make

• Es una utilidad para automatizar la construcción automática de programas ejecutables y librerías desde el código fuente.

• Es decir, se emplea para compilar código.

• Compila solo lo necesario.

• Los archivos llamados Makefiles contienen las reglas que la utilidad Make Ejecutará.

•Existen cuatro tipos de líneas en un Makefile:

- ✓ Asignaciones.
- ✓ Comentarios.
- ✓ Objetivos.
- ✓ Comandos.

Proceso de recompilación con make

- Cuando se escriben grandes programas, el proceso de recompilación de programas grandes toma mucho más tiempo que los programas pequeños como es lógico. Muchas veces el trabajo se enfoca en un solo archivo o función de todo un programa y el resto permanece intacto.

- El comando make permite manejar grandes programas o grupos de programas debido mantiene un registro de que porciones del programa han sido cambiadas desde la última compilación y solo compila dichas secciones.

3. PROCESO DE COMPILACIÓN: UNO O VARIOS ARCHIVOS

Una simple compilación

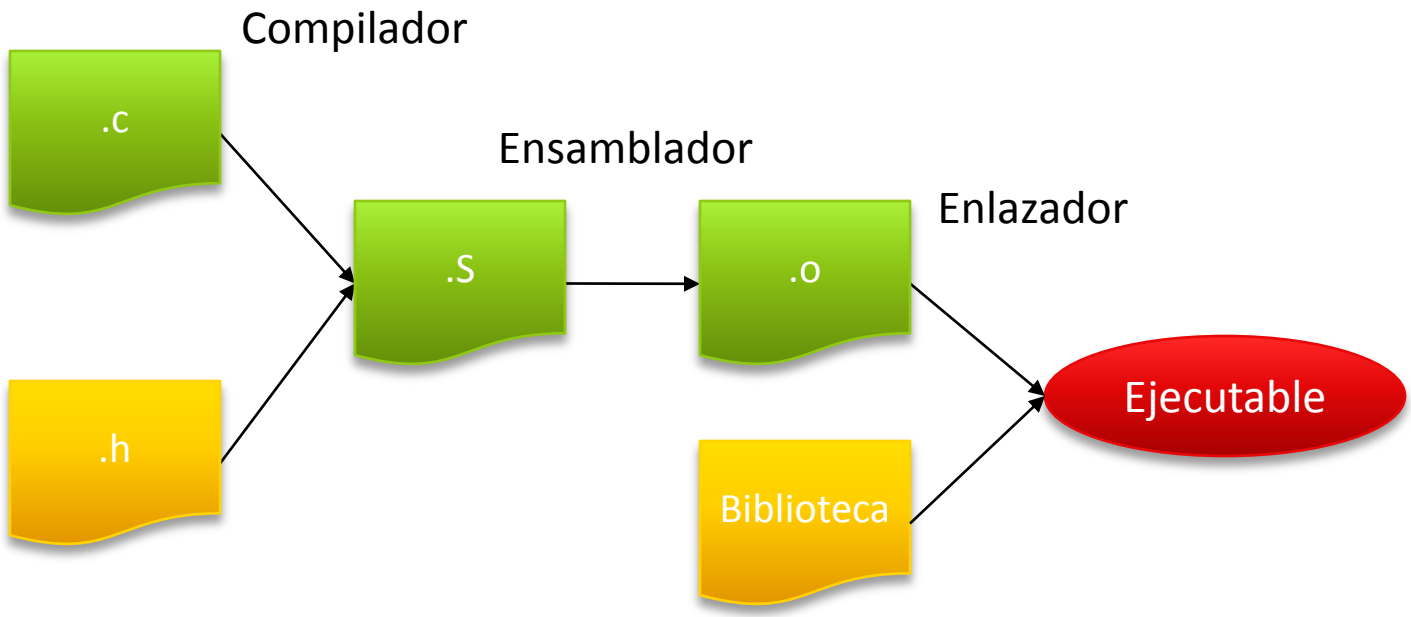
• Compilar un programa simple en C require al menos un archivo `.c` con archives de cabecera `.h`.

• El comando para compilar en este caso es simple `gcc archivo.c`.

• Hay tres pasos que se involucran en el proceso para obtener un programa ejecutable:

1. ***Etapa del compilador:*** Todo el código fuente C en el archivo `.c` es convertido en lenguaje ensamblador haciendo archivos `.S`.
2. ***Etapa del ensamblador:*** El código ensamblador generado en la etapa anterior es convertido en código objeto `.o`, que son fragmentos de código que el sistema entiende directamente.
3. ***Etapa del ensamblador:*** Es la etapa final donde se enlaza el código objeto del programa con el código de bibliotecas. Ésta etapa es la que genera un programa ejecutable.

Una simple compilación



Compilación con varios archivos

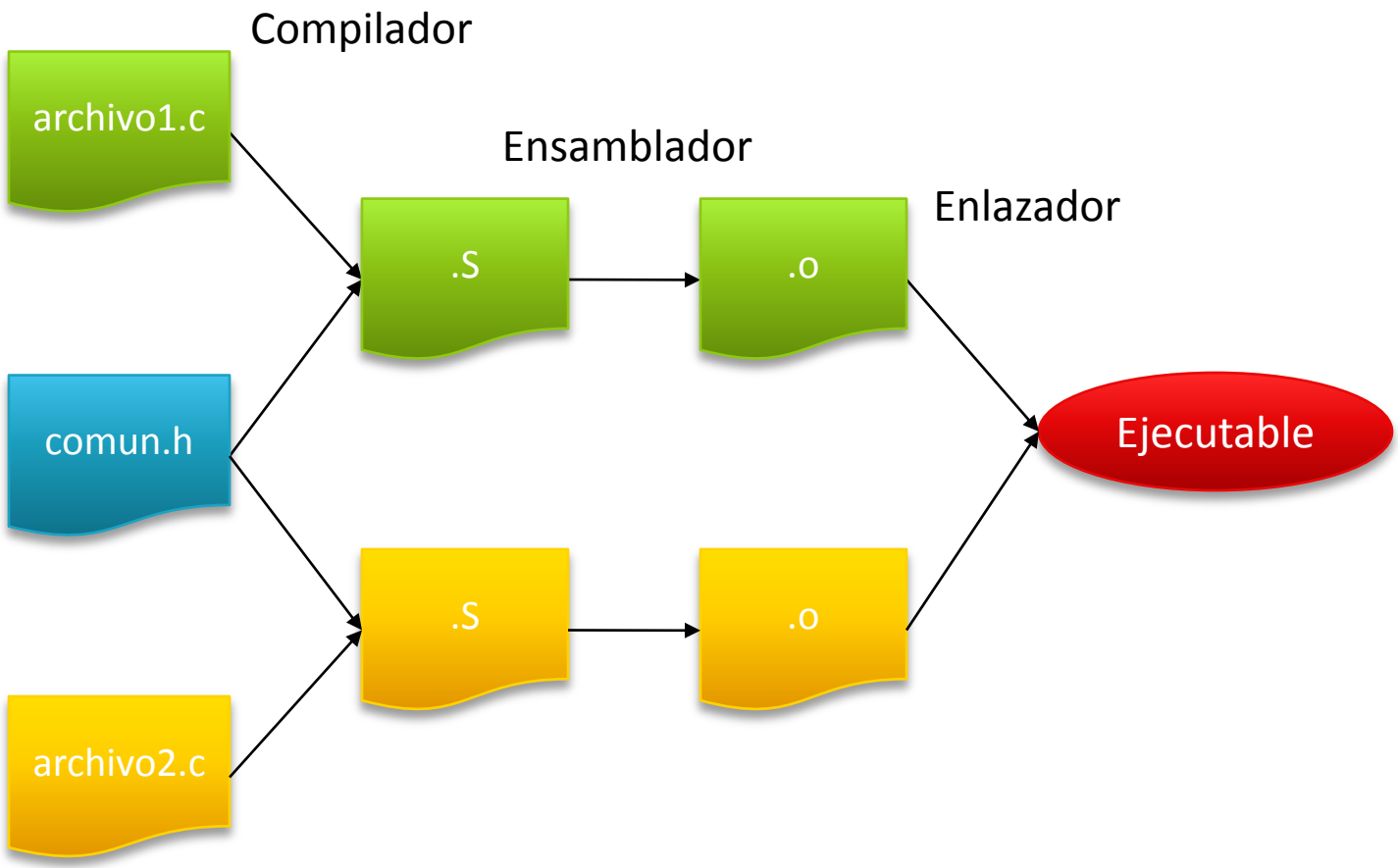
• Cuando un programa se hace muy grande, tiene sentido separar el código fuente en varios archivos `.c`.

• El comando que se emplea para compilar un programa con dos archivos y un `.h` común es:

```
gcc archivo1.c archivo2.c -o ejecutable
```

• Los dos primeros pasos son idénticos que cuando se compiló solo un archivo `.c`, pero el último paso tiene un giro interesante: los dos archivos objeto `.o` están enlazados juntos en la etapa de enlazado para crear un archivo ejecutable.

Compilación con varios archivos



Compilación con varios archivos

- Los pasos requeridos para crear el programa ejecutable pueden ser divididos en dos: el paso del compilador/ensamblador y el paso del enlazador. Los dos archivos objeto pueden ser creados por separado, pero ambos son requeridos en el paso final para crear el programa ejecutable.

- Se puede emplear la opción **-c** con **gcc** para crear el archivo objeto correspondiente. Así los tres pasos requeridos para generar el programa ejecutable a partir de archivos objeto separados son:

```
gcc -c archivo1.c
gcc -c archivo2.c
gcc archivo1.o archivo2.o -o ejecutable
```

Partiendo un programa C

• Cuando se parte un programa C en varios archivos se deben tener en cuenta los siguientes aspectos:

✓ Asegurarse que una función con el mismo nombre no esté definida dos archivos o el compilador se confundirá.

✓ De igual manera se debe asegurar que no hay variables globales duplicadas.

✓ Si se emplean variables globales procure definirlas en un solo archivos por ejemplo un `.h` como: `extern int variableGlobal;`.

✓ Para usar funciones de otro archivo, haga un archivo `.h` con los prototipos de funciones y use `#include` para incluir aquellos archivos .h en los archivos `.c`.

✓ Al menos uno de los archivos debe tener una función `main()`.

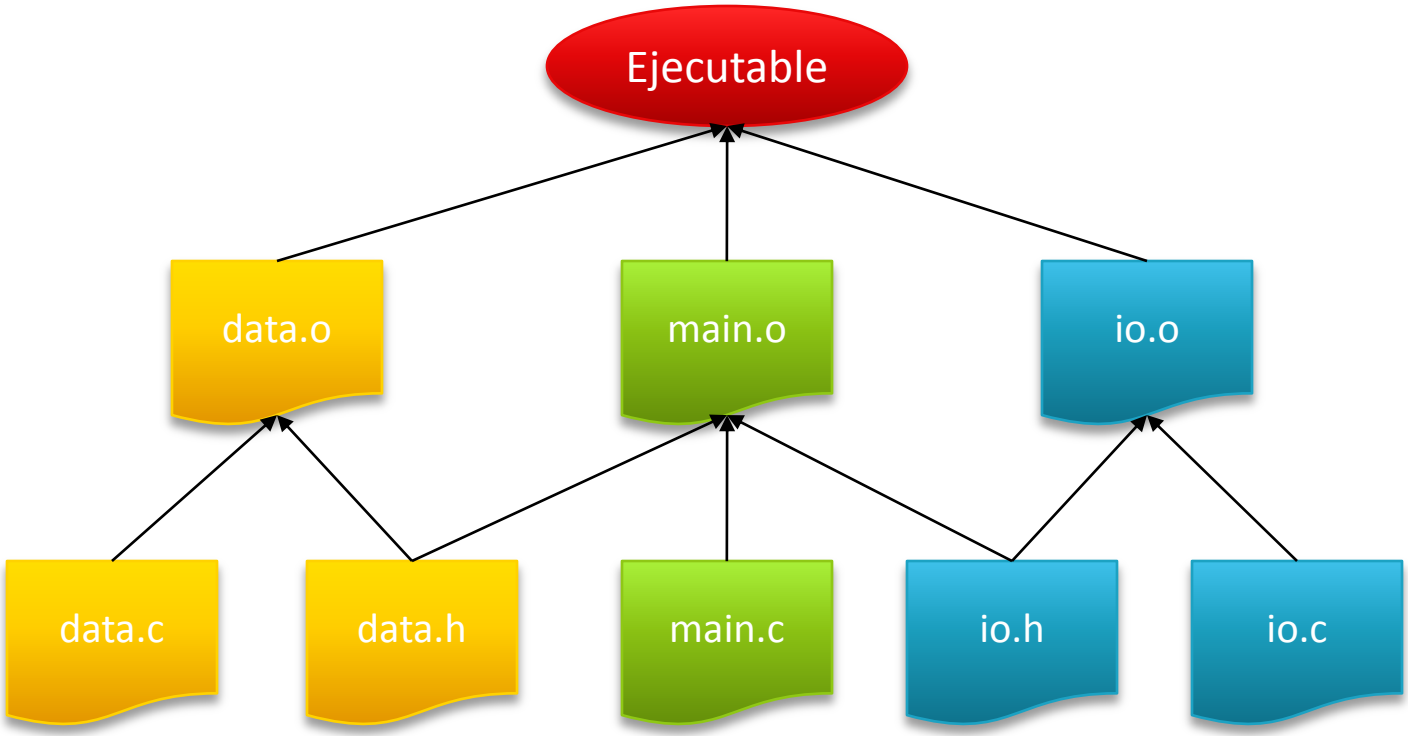
3. DEPENDENCIAS

- El principio por medio del cual **`make`** opera fue descrito en la sección anterior. Crea programas de acuerdo con las dependencias de archivos.

- Por ejemplo se sabe que con el objetivo de crear una archivo objeto, **`programa.o`** se requiere de al menos un archivo **`programa.c`** y probablemente otras dependencias como un archivo .h.

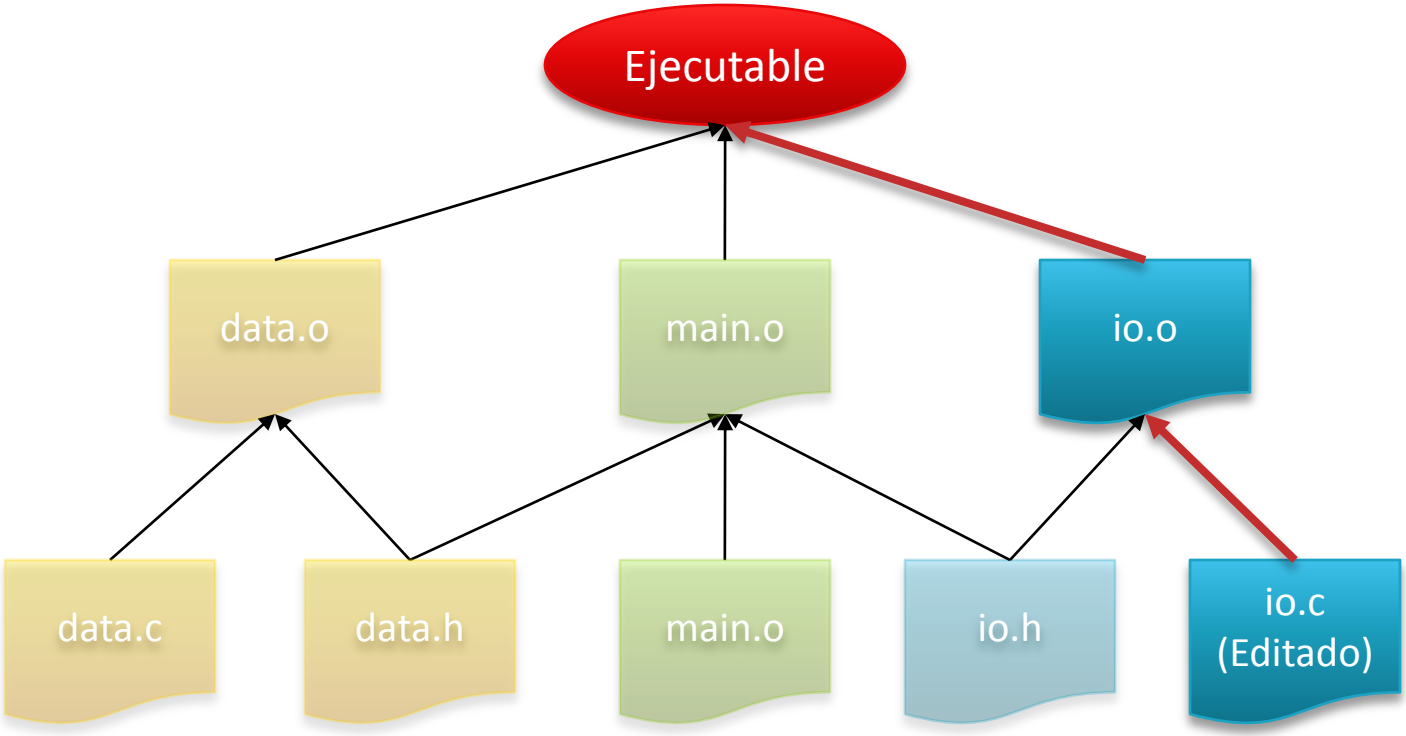
Gráficos de dependencias

El siguiente gráfico es un programa que está conformado por 5 archivos de código fuente, llamados, data.c, data.h, io.c, io.h y main.c. En el tope está el resultado final que es el programa ejecutable. Las líneas que salen de un archivo hacia abajo indican los archivos de los cuales dependen. Por ejemplo para crear main.o, tres archivos son necesarios data.h, io.h y main.c.



¿Como trabajan las dependencias?

- Supongamos que ya el proceso de compilación del programa fue realizado, pero mientras se revisa el programa se encuentra que una función en `io.c` tiene un error, entonces se edita dicha archivo para arreglar el error.
- Si se sube el gráfico a partir de `io.c` se nota que `io.o` necesita ser actualizado debido a que `io.c` cambio, de igual manera debido a que `io.o` cambió el archivo ejecutable debe ser actualizado.



¿Cómo es que make maneja las depencias?

• El programa make obtiene su “gráfico” de dependencias de un archivo llamado Makefile que reside en el mismo directorio donde se encuentra el código fuente. Make revisa el tiempo de la modificación de los archivos y cuando un archivo cambia ejecuta el compilador solo sobre dicho archivo, lo cual reduce el tiempo de compilación.

• En el ejemplo anterior io.c cambió de tal manera que se convierte en un archivo más nuevo que io.o, lo cual implica que make debe correr gcc –c io.c para crear un nuevo io.o y después ejecuta gcc data.o main.o io.o –o ejecutable para regenerar el ejecutable.

2. MAKEFILE

Reglas de un Makefile

• Una regla en un **Makefile** le dice a **Make** una serie de comandos necesarios con el objetivo de construir un archivo objetivo a partir de archivos fuente o bien realizar una acción cuya salida no es un archivo.

• Una regla se compone de tres partes:

✓Objetivo.
✓Comandos.
✓Pre-requisitos o dependencias.

```
objetivo: dependencias
<tab><tab>comando(s)
```

Objetivos

• Es usualmente el nombre del archivo que es generado por un programa. Usualmente estos archivos son archivos ejecutables u objetos.

• El siguiente es un ejemplo de un objetivo:

```
objetivo:
```

• Un objetivo también puede ser una acción que se realiza y no un archivo de salida. Éste tipo de objetivos se les conoce como objetivos **PHONY** y son uno o un conjunto de comandos que se ejecutan en secuencia sin generar un archivo de salida.

```
.PHONY clean
clean:
```

• Los objetivos no comienzan con tabulación o espacios.

Comandos

Es la acción que se lleva a cabo en una regla. Una regla puede tener más de un comando y empiezan con dos tabulaciones.

```
<TAB><TAB>comando(s)
```

Dependencias o pre-requisitos

Las dependencias son objetivos que son ejecutados antes del objetivo en sí mismo.

```
objetivo: dependencia1 dependencia2
```

Asignaciones y comentarios

• Por convención las asignaciones en los Makefiles son escritas en mayúscula.

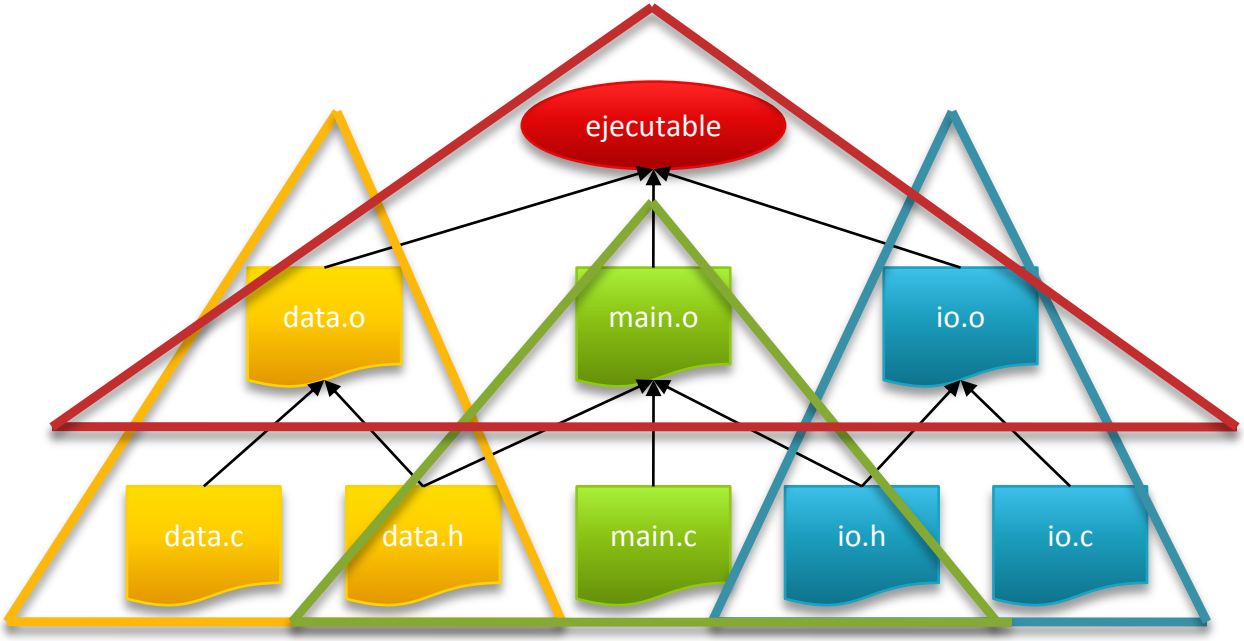
• Por ejemplo:

```
SALUDO= "Hola"
echo $SALUDO
```

• Los comentarios en los Makefiles comienzan con #.

```
# Comentario
```

Gráficos de dependencias y Makefiles




```
ejecutable: data.o main. io.o
<TAB><TAB>gcc data.o main.o io.o -o ejecutable
```

```
data.o: data.c data.h
<TAB><TAB>gcc -c data.c
```

```
main.o: main.c main.h
<TAB><TAB>gcc -c main.c
```

```
io.o: op.c io.h
<TAB><TAB>gcc -c io.c
```

Lista de dependencias

```
ejecutable: data.o main.o io.o
<TAB><TAB>gcc data.o main.o io.o -o ejecutable

data.o: data.c data.h
<TAB><TAB>gcc -c data.c

main.o: main.c main.h
<TAB><TAB>gcc -c main.c

io.o: op.c io.h
<TAB><TAB>gcc -c io.c
```

- Nótese que en el Makefile mostrado anteriormente, los archivos **.h** se incluyen en las dependencias, sin embargo, no se hace referencia a sus comandos correspondientes. Esto se debe a que los archivos **.h** están referenciados en el archivo **.c** correspondiente mediante **#include "archivo.h"**.

- Si no se incluye explícitamente los archivos **.h** en el Makefile el programa no será actualizado cuando se hagan cambios en dicho archivo.

Ejecutar un Makefile

- Make es capaz de reconocer tanto los archivos nombrados como **`Makefile`** o **`makefile`**. Sin embargo, si no se quiere usar dichos nombres se le puede indicar a Make cual es el archivo que debe utilizar **`make –f miMakefile`**.
- El orden en que las dependencias son listadas es importante. Si simplemente se digita **`make`**, se ejecutará la primera regla del archivo.

```
$ make
    cc -c data.c
    cc -c main.c
    cc -c io.c
    cc data.o main.o io.o -o project1
```

- También se puede especificar alguna de las reglas del Makefile si solo se requiere que dicha regla sea ejecutada.

```
$ make objetivo
```

4. OPTIMIZACIONES DE LOS MAKEFILE'S

Macros

• El Make permite el uso de macros que son similares a variables. El formato es el siguiente:

```
OBJS = data.o main.o io.o
```

• Cuando se requiera expandir las macros se emplea **$(OBJS)** . El siguiente es el ejemplo del **Makefile** usando macros:

```
OBJS = data.o main.o io.o
ejecutable: $(OBJS)
<TAB><TAB>gcc $(OBJS) -o ejecutable

data.o: data.c data.h
<TAB><TAB>gcc -c data.c

main.o: main.c main.h
<TAB><TAB>gcc -c main.c

io.o: op.c io.h
<TAB><TAB>gcc -c io.c
```

Macros

• También es posible especificar los valores de las macros cuando se ejecuta **make**, de la siguiente manera:

```
make 'OBJS=data.o main.o io_nuevo.o'
```

•Lo anterior sobreescribe el valor de **OBJS** en el **Makefile**.

• También es posible manipular la forma en que los macros son evaluados. Asumiendo que **OBJS = data.o io.o main.o**, al emplear **$(OBJS:.o=.c)**, dentro del **Makefile** se sustituye **.o** al final con **.c** lo cual da como resultado: **data.c main.c io.c**.

Macros especiales

• Adicional a las macros que el usuario crea, existen algunas macros predefinidas por **Make**. Aquí hay algunas de ellas:

✓ `$@` - Nombre complete del objetivo actual.

✓ `$?` - Una lista de los archivos para la dependencia actual los cuales están desactualizados.

✓ `$<` - El archivo de código fuente de la dependencia actual.

Macros del Toolchain

• Normalmente en los Makefiles se definen macros que hacen referencia a las herramientas del Toolchain. Los siguientes son ejemplos basados en un Cross-Toolchain de ARM.

```
# Nombres de herramientas
CROSS_COMPILE    = arm-linux-gnueabi-
AS               = $(CROSS_COMPILE)as
AR               = $(CROSS_COMPILE)ar
CC               = $(CROSS_COMPILE)gcc
CPP              = $(CC) -E
LD               = $(CROSS_COMPILE)ld
NM               = $(CROSS_COMPILE)nm
OBJCOPY          = $(CROSS_COMPILE)objcopy
OBJDUMP          = $(CROSS_COMPILE)objdump
RANLIB           = $(CROSS_COMPILE)ranlib
READELF          = $(CROSS_COMPILE)readelf
SIZE             = $(CROSS_COMPILE)size
STRINGS          = $(CROSS_COMPILE)strings
STRIP            = $(CROSS_COMPILE)strip

# Configuración de construcción
CFLAGS           = -O2 -Wall -I.
LDFLAGS          = -L.
```

Reglas predefinidas

El **Make** por sí mismo sabe que para crear un archive **.o**, debe usar **gcc -c** sobre el archive **.c** correspondiente. Éstas reglas ya existen en **Make** así que se puede tomar ventaja de eso para obtener un **Makefile** más pequeño debido a que no se incluye el comando del compilador.

```
OBJECTS = data.o main.o io.o

ejecutable: $(OBJECTS)
<TAB><TAB>gcc $(OBJECTS) -o ejecutable

data.o: data.h
main.o: data.h io.h
io.o: io.h
```

5. EJEMPLOS DE MAKEFILE'S

Ejemplo de un Makefile

```
PHONY: clean

APP=holamundo
SRC=$(APP).c
OBJ=$(APP).o
CC=gcc
CFLAGS = -c  -Wall
LFLAGS = -Wall

$(APP): $(OBJ)
<TAB><TAB>$(CC) $(CFLAGS) $(OBJ) -o $(APP)

$(OBJ): $(SRC)
<TAB><TAB>$(CC) $(LFLAGS) $(SRC)

clean:
<TAB><TAB>rm $(APP) $(OBJ)
```

Ejemplo de un Makefile

En el ejemplo anterior hay tres reglas:

1. Una que genera el archivo objeto producto de la compilación.

2. Una que enlaza el archivo objeto para generar el archivo ejecutable.

3. Una que limpia los archivos generados, la cual es tipo PHONY.

Algunas consideraciones

• Si en la consola solo se digita “make” se ejecutará la primera regla.

• Las dependencias de una regla deben ser declaradas después de la regla. En éste sentido el siguiente ejemplo es incorrecto:

```
$(OBJ): $(SRC)
<TAB><TAB>$(CC) $(CFLAGS) $(SRC)

$(APP): $(OBJ)
<TAB><TAB>$(CC) $(LFLAGS) $(OBJ) -o $(APP)
```

Compilación con archivos múltiples

```
PHONY: limpiar

APP=holamundo
SRCS=$(APP)1.c $(APP)2.c
OBJS=$(APP)1.o $(APP)2.o
CC=gcc
CFLAGS=-c -Wall
LFLAGS=-Wall

$(APP): $(OBJS)
<TAB><TAB>$(CC) $(CFLAGS) $(OBJS) -o $(APP)

$(OBJ): $(SRCS)
<TAB><TAB>$(CC) -c $(CFLAGS) $(SRCS)

clean:
<TAB><TAB>rm $(APP) $(OBJS)
```

Compilación con archivos múltiples simplificado

```
PHONY: limpiar

APP=holamundo
SRCS=$(APP)1.c $(APP)2.c
OBJS=$(APP)1.o $(APP)2.o
CC=gcc
CFLAGS=-c -Wall
LFLAGS=-Wall

$(APP): $(OBJS)
<TAB><TAB>$(CC) $(CFLAGS) $(OBJS) -o $(APP)

.c.o:
<TAB><TAB>$(CC) -c $(CFLAGS) $<

limpiar:
<TAB><TAB>rm $(APP) $(OBJS)
```