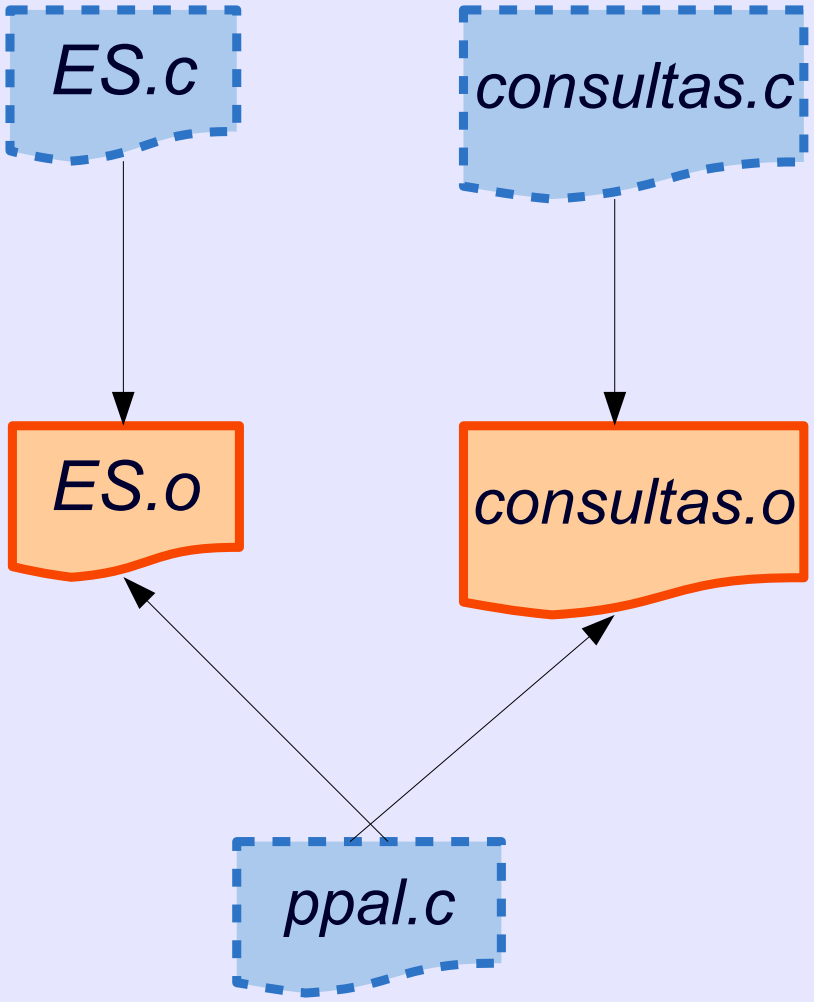
Tema 8: Herramientas

El programa *make* y la construcción de ficheros *makefiles*

Introducción

- Durante el proceso de desarrollo, el software se modifica frecuentemente y las modificaciones pueden afectar a otros módulos.
- Estas modificaciones deben propagarse a los módulos que dependen de aquellos que han sido modificados, para que el programa ejecutable final refleje las modificaciones introducidas.



Introducción

- **Herramienta Make**
  - Herramienta para hacer más eficiente la compilación y enlace (link) de los programas.
  - Recompila únicamente los módulos que se han modificado y los que depende de ellos.

- Necesitamos un fichero de descripción, llamado genéricamente ***makefile***, que contiene:
  - Las órdenes que debe ejecutar *make*
  - Las dependencias entre los distintos módulos del proyecto.

Introducción

Funcionamiento

- Examina la lista de dependencias y determina qué ficheros ha de construir:
  - Si el fichero no está creado, lo crea
  - Si el fichero está creado, mira la hora y la fecha asociados y, si el fichero fuente es más reciente que el fichero destino, lo reconstruye.
    - Este mecanismo hace posible mantener siempre actualizada la última versión.

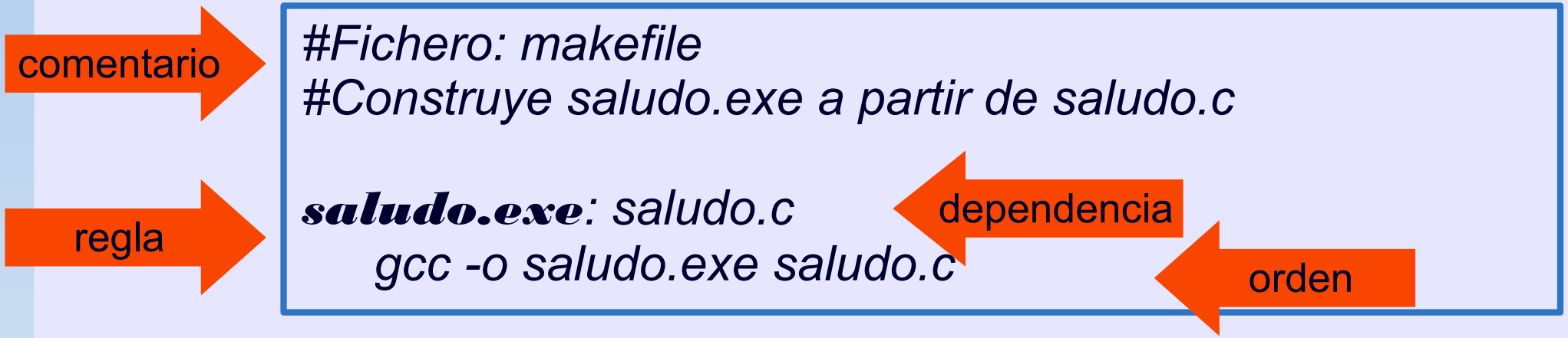
Introducción

- El uso de la herramienta *make* y los ficheros *makefiles* proporciona un mecanismo para realizar una gestión inteligente, sencilla y precisa de un proyecto de software (programa), ya que permite:
  - **Especificar las dependencias** entre los módulos de un proyecto de software.
  - **Recompilar** únicamente los módulos que han de actualizarse.
  - Obtener siempre la **última versión** que refleja las modificaciones realizadas.
  - Un mecanismo *casi estándar* de gestión de proyectos de software, independientemente de la plataforma en la que se desarrolla: puede variar ligeramente de unos sistemas a otros.

Ficheros *makefile*

- Un fichero *makefile* contiene:
  - Las órdenes que debe ejecutar la utilidad *make*
  - Las dependencias entre los distintos módulos del proyecto
- Elementos que aparecen en un fichero *makefile* son:
  - Comentarios
  - Reglas
    - Explícitas
    - Implícitas
  - Órdenes
  - Destinos simbólicos
  - Destinos phony
  - Macros

Ficheros *makefile*



- Comentarios
  - Tienen como objetivo clarificar el contenido del fichero *makefile*
  - Un comentario empieza por el símbolo #
  - Son comentarios de una línea
    - # Esto es un programa de ejemplo

Ficheros *makefile*

- Reglas
  - Mecanismos por los que se le indica a la utilidad *make*
    - Qué hacer (Los destinos)
    - A partir de qué (Las dependencias)
    - Cómo hacerlo (cómo construir los destinos)
  - Dos tipos de reglas
    - Las **reglas explícitas** dan instrucciones a make para que construya los ficheros especificados
    - Las **reglas implícitas** dan instrucciones generales que make sigue cuando no puede encontrar una regla explícita.
      - Para saber que reglas trae make predefinidas, se puede teclear:
        - make -p

Ficheros *makefile*

- Un fichero *makefile* está compuesto por un conjunto de reglas.
- El formato habitual de una regla explícita es el siguiente:

```
destino:lista de dependencias
        orden(es)
```

  - *Destino*: fichero destino a construir
  - *Lista de dependencias:* ficheros de los que depende el destino
    - Se especifican los nombres de los archivos separados por espacios en blanco
  - *orden(es):* órdenes válidas para el sistema operativo en que se ejecuta el make. Usualmente sirven para construir el destino, aunque no tiene porque ser así

Fichero *makefile*

- MUY IMPORTANTE:
  - Cada línea de órdenes empezará con un TABULADOR.
  - Si no es así, make mostrará un error y no continuará procesando el fichero makefile

destino: lista de dependencias
orden(es)

Tabulador

Sintaxis programa *make*

- ***make [opciones] [destinos]***
  - Cada ***opción*** va precedida por un signo “-” o una barra inclinada /
  - ***destino*** indica el destino que debe crear. Generalmente se trata del fichero que debe crear o actualizar, estando especificado en el fichero makefile el procedimiento de creación/actualización del mismo
- Las **opciones** más frecuentes son:
  - ***-h o -help***: proporciona ayuda a cerca del main.
  - ***-f fichero:*** indica cómo se llama el fichero (en vez de makefile o Makefile) que queremos que ejecute la orden make.
  - ***-n, --just, --dry***: muestra las instrucciones que ejecutaría la utilidad make, pero no las ejecuta. Se usa para verificar la corrección del makefile.

Funcionamiento de *make*

- En primer lugar, busca el fichero makefile que debe interpretar:
  - Si se ha especificado la opción -f fichero, busca ese fichero
  - Si no, busca en el directorio actual un fichero llamado makefile o Makefile.
    - Si lo encuentra, lo interpreta
    - Si no, da un mensaje de error y termina
- Intenta construir el(los) destino(s) especificado(s)
  - Si no se proporciona ningún destino, sólo construye el primer destino que encuentre en el fichero makefile.
  - Para construir un destino, es posible que deba construir antes otros destinos. Si es así, los construye examinando la lista de dependencias (dependencia encadenada).

Funcionamiento de *make*

- Si falla al construir algún destino en cualquier paso:
  - Se detiene la ejecución.
  - Muestra un mensaje de error.
  - Borra el destino que estaba construyendo.
  - Borra los destinos de la lista de dependencias.

Ejemplo 1: *makefile*

- *Makefile que crea un ejecutable a partir de un único fichero fuente.*

```
#Fichero: makefile
#Construye saludo.exe a partir de saludo.c

saludo.exe: saludo.c
    gcc -o saludo.exe saludo.c
```

- Como hay una única regla y el fichero se llama makefile, las siguiente órdenes tienen el mismo efecto:
  - make, make -f makefile, make saludo.exe,
  - make -f makefile saludo.exe

Ejemplo1: makefile

- Se incluyen dos líneas de comentarios al principio del fichero *makefile* que indican las tareas que realizará la utilidad *make*
- En el ejemplo encontramos una única regla que indica que, para construir el destino *saludo.exe,* se requiere la existencia de *saludo.c*. Ese destino se construye ejecutando la orden
  - "*gcc -o saludo.exe saludo.c*"
  - Compila el fichero saludo.c y lo enlaza con las bibliotecas adecuadas para generar finalmente saludo.exe.

Ejemplo 2: *makefil2.mak*

- *Crea un ejecutable a partir de tres ficheros fuentes*

```
#Fichero: makefil2.mak

main.x: main.o vector.o orden.o
    gcc -o main.x main.o vector.o orden.o

main.o: main.c vector.h orden.h
    gcc -c main.c

vector.o: vector.c vector.h
    gcc -c vector.c

orden.o: orden.c orden.h
    gcc -c orden.c
```

Ejemplo 2: *makefil2.mak*

- Este ejemplo crea el destino main.x a partir de tres ficheros objetos (main.o, vector.o, orden.o)
  - Utiliza la orden gcc -o main.x main.o vector.o orden.o

- Si alguno de los tres ficheros no existe
  - Busca una regla en el fichero makefile que permita construirlo
  - Reglas que construyen un fichero objeto a partir del fichero fuente:
    - *gcc -c main.c*
    - *gcc -c vector.c*
    - *gcc -c orden.c*

Ejemplo 2: *makefil2.mak*

- Orden para construir el primer destino (ejecutable)
  - *make -f makefil2.mak*
  - *make -f makefil2.mak main.x*
- Órdenes para construir algún destino .o
  - *make -f makefil2.mak main.o*
  - *make -f makefil2.mak vector.o*
  - *make -f makefil2.mak orden.o*

Ejemplo 3: makefil3.mak

- *Crea un ejecutable a partir de tres ficheros fuentes, usando una biblioteca.*

```
main.x: main.o lib.a
    gcc -o main.x main.o lib.a

main.o: main.c vector.h orden.h
    gcc -c main.c

lib.a: vector.o orden.o
    ar -rsv lib.a vector.o orden.o

vector.o: vector.c vector.h
    gcc -c vector.c

orden.o: orden.c orden.h
    gcc -c orden.c
```

Ejemplo 3: makefil3.mak

- Orden para construir el primer destino (ejecutable)
  - *make -f makefil3.mak*
  - *make -f makefil3.mak main.x*
- Órdenes para construir el resto de destinos
  - *make -f makefil3.mak main.o*
  - *make -f makefil3.mak vector.o*
  - *make -f makefil3.mak orden.o*
  - *make -f makefil3.mak lib.a*

Órdenes

- Se puede incluir cualquier orden válida del sistema operativo en el que se ejecute la utilidad *make*.
- Pueden incluirse cuantas órdenes se requieran como parte de una regla, **cada una en una línea distinta**.
- Las órdenes pueden ir precedidas por **prefijos:**
  - @ Desactivar el eco durante la ejecución de esa orden
  - - Ignorar los errores que puede producir la orden a la que precede
- Es **imprescindible** que cada línea de órdenes empiece con un **tabulador.**

Órdenes

- Se puede incluir cualquier orden válida del sistema operativo en el que se ejecute la utilidad *make*.
- Pueden incluirse cuantas órdenes se requieran como parte de una regla, **cada una en una línea distinta**.
- Las órdenes pueden ir precedidas por **prefijos:**
  - @ Desactivar el eco durante la ejecución de esa orden
  - - Ignorar los errores que puede producir la orden a la que precede
- Es **imprescindible** que cada línea de órdenes empiece con un **tabulador**

Ejemplo 4: *makefil4.mak*

- *Makefile que incorpora una regla sin lista de dependencias*

```
saludo.exe: saludo.c
    echo Creando saludo.exe
    gcc -o saludo.exe saludo.c

#Esta regla especifica un destino que no es
#un ejecutable
saludo.o: saludo.c
    @echo Creando saludo.o solamente
    gcc -c saludo.c

#Esta regla especifica un destino sin lista de
#dependencias
clean:
    @echo Borrando ficheros.o
    @rm *.o
```

La orden *clean* sólo la incluiremos si realmente queremos borrar los ficheros .o de nuestro proyecto

Ejemplo 4: *makefil4.mak*

- La regla cuyo destino es clean no tiene asociada una lista de dependencias, pues su construcción no requiere de la construcción de otro destino previo. Basta ejecutar:

  make -f makefil4.mak clean

- La orden que tiene asociada (*rm *.o*) lo que hace es borrar todos los archivos que tenga extensión .o

- Si se ejecuta *make -f makefil4.mak*, se visualiza:

  Creando saludo.exe...

  gcc -o saludo.exe saludo.c

Ejemplo 4: *makefil4.mak*

- Por defecto, se muestran en consola las órdenes que se van ejecutando.
- Si no se hubiera usado el prefijo @ delante de la orden echo, el resultado hubiera sido:

  echo Creando saludo.exe ...

  Creando saludo.exe ...

  gcc -o saludo.exe saludo.c

- Se puede incluso poner el prefijo @ delante de la llamada a gcc y el resultado sería:

  Creando saludo.exe ...

Destinos simbólicos

- Permiten construir varios destinos sin necesidad de invocar a la herramienta *make* tantas veces como destinos se deseen construir
- Un destino simbólico se especifica en un fichero makefile en la **primera línea** operativa del mismo.
- En su sintaxis se asemeja a la especificación de una regla, con la diferencia de que **no tiene asociada ninguna orden**.
- El formato es el siguiente:

  destinoSimbolico: *listas de destinos*

- donde:
  - *destinoSimbolico* es el nombre del destino simbólico
  - *lista de destinos:* especifica los destinos que se construirán cuando se invoque al make

Destinos simbólicos

- Al estar en la primera línea operativa del fichero makefile, *make*:
  - Intenta construir el **destino simbólico**.
    - Examina la lista de destinos y construye cada uno de ellos
    - Debe existir una regla para cada uno de los destinos
  - Finalmente intenta construir el **destino simbólico**
    - Como no habrá ninguna instrucción que le indique cómo construirlo, *make* no hará nada más.
    - El objetivo está cumplido: se han construidos varios destinos con una sola ejecución de *make*.
- El nombre dado al destino simbólico no tiene importancia.

Ejemplo 5: *makefil5.mak*

- *Makefile con destino simbolico llamado saludos*

```
#Fichero: makefil5.mak

saludos: saludo.exe saludo2.exe clean

saludo.exe: saludo.c
	@echo Creando saludo.exe
	gcc -o saludo.exe saludo.c

saludo2.exe: saludo2.c
	@echo Creando saludo2.exe
	gcc -o saludo2.exe saludo.c

clean:
	@echo Borrando ficheros.o ...
	@rm *.o
```

Ejemplo 5: *makefil5.mak*

- Si no se especifica ningún destino, *make* intentará construir el primero, llamado *saludos*.

- Antes debe construir (si no lo están) todos los que aparecen en la lista de dependencia asociada: *saludo.exe, saludo2.exe, clean*

- Una vez construidos, se plantea la construcción de *saludos*, pero al no tener ningún orden para ello, termina.

Destinos phony

- Si existe en el directorio actual un fichero con el nombre clean, la orden "*make -f makefil5.mak clean*" no borrará los ficheros
  - El destino *clean* no tiene ninguna dependencia y existe un fichero llamado *clean*. La herramienta make sobreentiende que ese destino no es necesario reconstruirlo.
- SOLUCIÓN: Declarar este tipo de destinos como *falsos (PHONY)* de la siguiente forma:

.PHONY: clean

  - Esta regla se puede poner en cualquier parte del fichero makefile, normalmente antes de la regla en cuestión.
  - Con esto, al ejecutar *make -f makefil5.mak clean* todo funcionaría bien, aunque exista un fichero llamado clean.

Ejemplo 6: *makefil6.mak*

- *Makefile con destino PHONY*

```
#Fichero: makefil6.mak
saludos: saludo.exe saludo2.exe clean

saludo.exe: saludo.c
	@echo Creando saludo.exe
	gcc -o saludo.exe saludo.c

saludo2.exe: saludo2.c
	@echo Creando saludo2.exe
	gcc -o saludo2.exe saludo.c

.PHONY: clean
clean:
	@echo Borrando ficheros.o ...
	@rm *.o
```

Makefile tipo

```
all: main.x clean

main.x: main.o lib.a
	gcc -o main.x main.o lib.a

main.o: main.c vector.h orden.h
	gcc -c main.c

lib.a: vector.o orden.o
	ar -rsv lib.a vector.o orden.o

vector.o: vector.c vector.h
	gcc -c vector.c

orden.o: orden.c orden.h
	gcc -c orden.c

.PHONY: clean
clean:
	@echo Borrando ficheros.o ...
	@rm *.o
```

Complementario

Macros en ficheros *makefiles*

- Una macro o variable es una cadena que se expande cuando se ejecuta un fichero *makefile*.
- Las macros permiten crear ficheros *makefile* genéricos.
- Una macro puede representar:
  - Lista de nombres de ficheros.
  - Opciones del compilador.
  - Programas a ejecutar.
  - Directorios donde buscar los ficheros fuentes.
  - Directorios donde escribir la salida, etc.

Macros en ficheros *makefiles*

- Se pueden utilizar dos tipos de macros:
  - Predefinidas (ya están en el sistema)
  - Definidas por el usuario

- Predefinidas (ya están en el sistema)
  - `$^`: equivale a *todas* las dependencias de la regla, con un espacio entre ellas.
  - `$<`: nombre de la *primera dependencia* de la regla.
  - `$@`: nombre del fichero *destino* de la regla.

Ejemplo 7: *makefil7.mak*

- *Makefile con macros predefinidas*

```
#Fichero: makefil7.mak

destinos: main.x clean

main.x: main.o vector.o orden.o
	gcc -o $@ $^

main.o: main.c vector.h orden.h
	gcc -c main.c
vector.o: vector.c vector.h
	gcc -c vector.c
orden.o: orden.c orden.h
	gcc -c orden.c
clean:
	rm *.o
```

Ejemplo 7: *makefil7.mak*

- Cuando *make* procesa *makefil7.mak* sustituirá la macro $@ y $^ por su valores correspondientes.
- La regla

```
main.x: main.o vector.o orden.o
        gcc -o $@ $^
```

- La procesa como

```
main.x: main.o vector.o orden.o
        gcc -o main.x main.o vector.o orden.o
```

- La macro $@ se sutituye por main.x (destino de la regla).
- y $^ se sustituye por la lista de dependencias (main.o vector.o orden.o).

Macros en ficheros *makefiles*

- Definidas por el usuario
  - Sintaxis

    Nombre = texto a expandir
  - donde
    - **Nombre** es el nombre de la macro. Es sensible a las mayúsculas y no puede contener espacios en blanco. La costumbre es utilizar nombres en **mayúscula**.
    - **Texto a expandir** es una cadena que puede contener cualquier carácter alfanumérico, de puntuación o espacios en blanco.
- Cada macro debe estar en **una línea separada** en el fichero makefile y se sitúan, normalmente, al principio de éste.

Macros en ficheros *makefiles*

- Si make encuentra más de una definición para el mismo **Nombre** (no es habitual), l**a nueva definición reemplaza a la antigua**.
- Para expandir una macro escribimos $(NOMBRE).
- La expansión de la macro se hace recursivamente. Si la macro contiene referencias a otras macros, estas referencias serán expandidas también.

```
DEBUG = -g
CFLAGS = $(DEBUG) -c
```

CFLAGS se expandirá a “-g -c”

Ejemplo 8: *makefil8.mak*

- *Makefile con macros definidas por el usuario*

```
#Fichero: makefil8.mak
OBJ = main.o vector.o orden.o
FLAGS = -g -c

destinos: main.x clean
main.x: $(OBJ)
	gcc -o main.x $(OBJ)
main.o: main.c orden.h vector.h
	gcc $(FLAGS) main.c
vector.o: vector.c vector.h
	gcc $(FLAGS) vector.c
orden.o: orden.c orden.h
	gcc $(FLAGS) orden.c
clean:
	rm $(OBJ)
```

Ejemplo 8: *makefil8.mak*

- En el ejemplo se define, al principio de las líneas operativas del fichero, la macro llamada **OBJ** cuyo valor es la cadena "*main.o vector.o orden.o*".
- Cuando *make* procesa *makefil8.mak* sustituirá las apariciones de $(OBJ) por el valor de la macro OBJ.
- La regla

```
main.x: $(OBJ)
    gcc -o main.x $(OBJ)
```

- la procesa como

```
main.x: main.o vector.o orden.o
    gcc -o main.x main.o vector.o orden.o
```

Ejemplo 8: *makefil8.mak*

- Se define otra macro llamada **FLAGS** a la que se le asigna la cadena "*-g -c* " (opciones para compilar).
- La regla

  orden.o: orden.c

  gcc $(FLAGS) orden.c
- la procesa como
  - *orden.o: orden.c*
    - *gcc -g -c orden.c*

Directivas condicionales en makefiles

- Se parecen a las directivas condicionales del preprocesador de C.
  - ifdef
  - ifndef
  - ifeq
  - Ifneq
  - else
  - endif
- Permiten a make dirigir el flujo de procesamiento en un fichero makefile en función de la evaluación de una condición en una directiva condicional.

Directivas condicionales en makefiles

- Las directivas condicionales para ficheros makefiles son:
  - **ifdef *MACRO***: actúa como la directiva #ifdef de C pero con macros en lugar de directivas #define
  - **ifndef *MACRO***: actúa como la directiva #ifndef de C pero con macros en lugar de directivas #define
  - **ifeq(*arg1*,*arg2*)**. Devuelve verdad si los dos argumentos expandidos son iguales
  - **ifneq(*arg1*,*arg2*)**. Devuelve verdad si los dos argumentos expandidos son distintos.
  - **else**. Actúa como un else de C
  - **endif**. Termina una declaración ifdef, ifndef, ifeq, ifneq.

Ejemplo 8: *makefil8.mak*

- *Makefile con macros definidas por el usuario*

```
#SISTEMA = _WINDOWS_
#SISTEMA = _LINUX_

ejecutable.exe: linuxWin.c
ifndef SISTEMA
	@echo Error: constante SISTEMA sin definir
	@echo "utilice: make -f linuxWin.mak [ SISTEMA
=_WINDOWS_ | SISTEMA =_LINUX_]"
else
	gcc -D$(SISTEMA) linuxWin.c -o ejecutable.exe
endif
```