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

Eva Lucrecia Gibaja Galindo

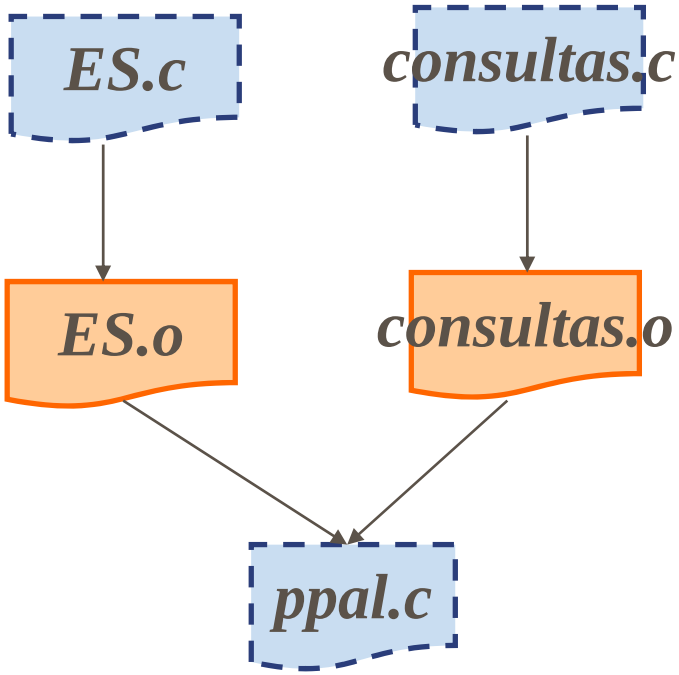
Dpto. Informática y Análisis Numérico

http://www.gnu.org/software/make/manual/html_chapter/make.html#SEC_Top

Introducción

- Durante el proceso de desarrollo el software se modifica frecuentemente y las modificaciones incorporadas 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

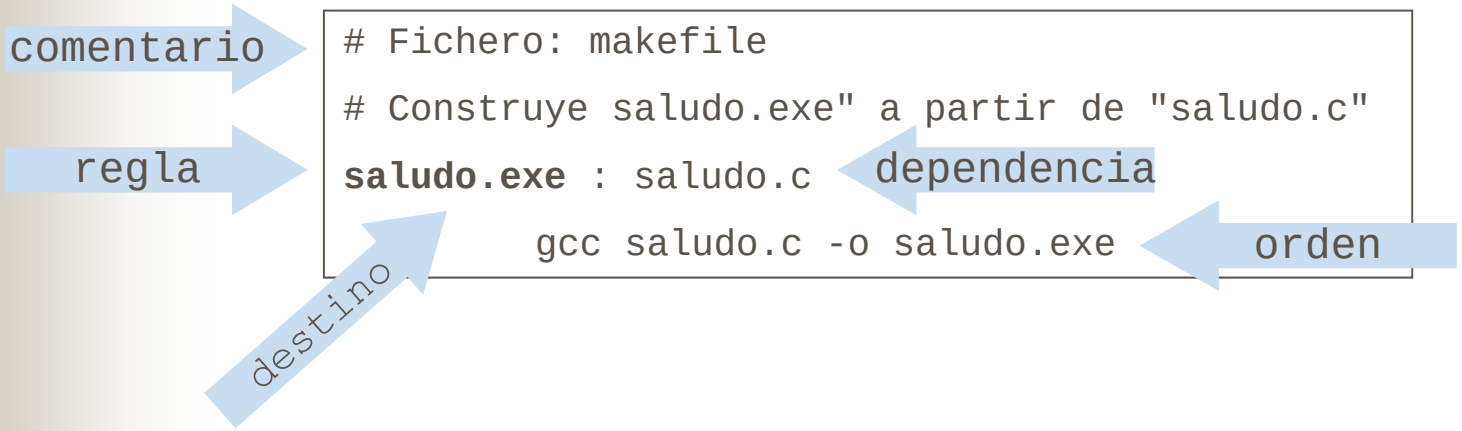
- El uso de la utilidad *make* y ficheros *makefile* proporciona el mecanismo para realizar una gestión inteligente, sencilla y precisa de un proyecto software, ya que permite:
  1. **Especificar la dependencia** entre los módulos de un proyecto software.
  2. **Recompilar** únicamente de los módulos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

Introducción

- La utilización de la orden *make* exige la creación previa de un fichero de descripción llamado genéricamente *makefile*, que contiene:
  - Las **órdenes** que debe ejecutar *make*.
  - Las **dependencias** entre los distintos módulos del proyecto.
- Examinando las listas de dependencia determina qué ficheros ha de reconstruir comparando la fecha y hora asociada a cada fichero:
  - Si el fichero fuente es más reciente que el fichero destino lo reconstruye.
  - Este mecanismo hace posible mantener siempre actualizada la última versión.

Ficheros *makefile*

- Los elementos de un fichero *makefile* son los siguientes:
  1. Comentarios.
  2. Reglas.
     - Reglas explícitas.
     - Reglas implícitas.
  3. Órdenes.
  4. Destinos simbólicos.



Comentarios

- Los comentarios tienen como objeto clarificar el contenido del fichero *makefile*.
- Una línea del comentario tiene **en su primera columna** el símbolo #.
- Los comentarios tienen el ámbito de **una línea**.

Reglas. Reglas explícitas

- Mecanismo por el que se indica a la utilidad *make:*
  - Los destinos.
  - Las listas de dependencias.
  - 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 qué reglas trae *make* predefinidas:

```
make -p
```

Reglas. Reglas explícitas

- El formato habitual de una regla explícita es el siguiente:

  destino: ***lista de dependencia***

  ordenes(es)

- **destino**: especifica el fichero a crear.
- ***lista de dependencia:*** ficheros de los que depende destino.
  - Se especifican los nombres de ficheros separados con espacios en blanco.
  - Si algún fichero especificado en esta lista se ha modificado, se busca una regla que contenga a ese fichero como destino y se construye.
  - Una vez construidas las últimas versiones de los ficheros especificados en ***lista de dependencia*** se construye **destino**.
- ***órden(es)***: órdenes válidas para el sistema operativo en el que se ejecute la utilidad *make*. Usualmente sirven para construir el destino, aunque no tiene porque ser así.

Reglas. Reglas explícitas

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.

Sintaxis del programa *make*

- ***make* [*opciones*] [*destino(s)*]**
  - 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 frecuentemente empleadas son las siguientes:
  - **-h** ó ***–help***: Proporciona ayuda acerca de *make*.
  - **-f *fichero***. Utilizaremos esta opción si se proporciona a *make* fichero makefile que no se llame *makefile* ni *Makefile*. Se toma el fichero llamado *fichero* como el fichero *makefile*.
  - ***NombreMacro[=cadena]*** define una constante simbólica (nombreMacro) con el nombre especificado como la cadena indicada después del signo =.
  - **-n , --just, --dry**: Muestra las instrucciones que *ejecutaría* la utilidad make, pero **no** las ejecuta. Para verificar la corrección del makefile.

Funcionamiento de *make*

1. 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* ó *Makefile*. Si lo encuentra, lo interpreta.
   - Si no, da un mensaje de error y termina.
2. Intenta **construir** el(los) destino(s) especificado(s).
   - Si no se proporciona ningún destino, intenta construir ***solo*** el primer destino que aparece en el fichero makefile.
   - Para construir un destino es posible que deba construir antes otros destinos. Si es así, los construye examinando las listas de dependencias. Esta reacción en cadena se llama **dependencia encadenada**.
3. Si en cualquier paso **falla** al construir algún destino:
   - Se detiene la ejecución.
   - Muestra un mensaje de error.
   - Borra el destino que estaba construyendo.
   - Borra los destinos de la lista de dependencias.

Ejemplo:makefile

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

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

Ejemplo:makefile

- Se incluyen dos líneas de comentario 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 saludo.c -o saludo.exe`" que compila el fichero *saludo.c* generando un fichero objeto temporal y lo enlaza con las bibliotecas adecuadas para generar finalmente *saludo.exe*.

Ejemplo:makefil2.mak

```
# Fichero: makefil2.mak
# Por defecto, Construye saludo.exe a partir de saludo.c
# Tambien puede construirse saludo.o a partir de saludo.c
saludo.exe : saludo.o
    gcc saludo.o -o saludo.exe
# Esta regla especifica un destino que no es un ejecutable
saludo.o : saludo.c
    gcc -c saludo.c -o saludo.o
```

Ejemplo:makefil2.mak

- En este ejemplo se especifican dos reglas:
  - La primera es la misma regla del ejemplo anterior.
  - La segunda especifica un destino que no es un fichero ejecutable.
- Estas dos órdenes tendrán el mismo efecto ya que *saludo.exe* es el primer destino de *makefil2.mak*.
  - `make -f makefil2.mak`
  - `make -f makefil2.mak saludo.exe`
- Para construir el segundo destino se ejecutará:
  - `make -f makefil2.mak saludo.o`

Ordenes

- 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
- Recordemos que es **imprescindible** que cada línea de órdenes empiece con un **tabulador**.

Ejemplo:makefil4.mak

```
# Fichero: makefil4.mak
# Por defecto, construye el saludo.exe a partir de saludo.c
# Incorpora dos reglas mas:
#   1) Crear el objeto saludo.o a partir de saludo.c
#   2) Novedad: Regla sin lista de dependencia.
saludo.exe : saludo.c
    echo Creando saludo.exe...
    gcc saludo.c -o saludo.exe

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

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

Ejemplo:makefil4.mak

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

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

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

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

Ejemplo:makefil4.mak

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

```
Creando saludo.exe...
```

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

```
make -f makefil4.mak clean
```

Destinos simbólicos

- Su finalidad es poder construir varios destinos sin necesidad de invocar a *make* tantas veces como destinos se desee 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 que **no tiene asociada ninguna orden**. El formato es el siguiente:

destino simbólico: *lista de destinos*

- Donde:
  - **destino simbólico** es el nombre del destino simbólico.
  - ***lista de destinos*** especifica los destinos que se construirán cuando se invoque a *make*.

Destinos simbólicos

- Al estar en la primera línea operativa del fichero makefile, *make:*
  1. 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.
  2. 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 construido varios destinos con una sola ejecución de `make`.
- El nombre dado al destino simbólico no tiene importancia.

Destinos phony

- Si existe en el directorio actual un fichero con el nombre `clean`, la orden "`make -f makefil6.mak clean`" no borrará los ficheros.
  - El destino `clean` no tiene ninguna dependencia y existe el fichero `clean`.
- SOLUCION: 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 `clean`.
  - Con esto, al ejecutar "*`make -f makefil6.mak clean`*" todo funcionará bien, aunque exista un fichero llamado `clean`
- Observar que también sería conveniente hacer lo mismo para el destino simbólico `saludos`.

Ejemplo:makefil6.mak

```
# Fichero: makefil6.mak
# Fichero makefile con un destino simbolico llamado "saludos"
saludos: saludo.exe saludo2.exe saludo3.exe clean
saludo.exe : saludo.c
    @echo Creando saludo.exe...
    gcc saludo.c -o saludo.exe
saludo2.exe : saludo2.c
    @echo Creando saludo2.exe...
    gcc saludo2.c -o saludo2.exe
saludo3.exe : saludo3.c
    @echo Creando saludo3.exe...
    gcc saludo3.c -o saludo3.exe
.PHONY: clean
clean :
    @echo Borrando ficheros .o...
    del *.o
```

Ejemplo:makefil6.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` y `saludo3.exe`.
- Una vez construidos, se plantea la construcción de `saludos`, pero al no tener ninguna orden para ello, termina.
- Se incorpora `clean` a la lista de dependencia del destino simbólico `saludos.`

Ejemplo de gestión de proyecto

- Rellenar un vector de 25 enteros con números aleatorios y ordenarlos (de forma ascendente) mediante el método de la burbuja. El programa `ordena1` utilizará tres funciones:
  1. **`llena_vector()`**, que rellena el vector con números aleatorios,
  2. **`pinta_vector()`**, que muestra el contenido del vector de forma tabulada, y
  3. **`ordena_vector()`**, que ordena *in situ* el vector: la secuencia ordenada estará en el mismo vector. El método de ordenación utilizado es el conocido como método de la ``burbuja''. Se trata del método más simple de la familia de métodos de ordenación basados en intercambio.

Ejemplo:version1-makefile.v1

- En la primera versión del programa, el código se presenta en un solo fichero fuente (*ordena1.c*).
- El destino simbólico llamado destinos hace que se genere únicamente el ejecutable *ordena1.exe*.

```
# Fichero: makefile.v1
destinos: ordena1.exe
ordena1.exe : ordena1.c
   gcc ordena1.c -o ordena1.exe
```

ordenacion/v1/makefile.v1.mak

Ejemplo:version2-makefile.v2

- En esta versión del programa, se estructura el código en dos ficheros fuente y en un fichero de cabecera.
  1. **ppal.c**:contiene únicamente la función main().
  2. **funcsvec.h**:
     - Definición de la constante MAX (la única que necesita conocer la función main())
     - Prototipos de las funciones llena_vector(), pinta_vector() y ordena_vector(), que son las que invoca main().
  3. **funcsvec.c**:
     - La función swap(). *main()* no necesita conocer esta función. Es **local** a *funcsvec.c*
     - Las definiciones de las constantes MAX_LINE y MY_MAX_RAND. Estas constantes se usan únicamente por las funciones definidas en este módulo, de ahí que se oculten a la función main() haciéndolas locales a este módulo.

ordenacion/v2/makefile.v2.mak

Ejemplo:version2-makefile.v2

```
/*****************************************************************/
// Fichero: ppal.c
/*****************************************************************/
#include <stdio.h>
#include "funcsvec.h “

int main (void)
{
   int m [MAX];       // vector de trabajo. MAX definido en ppal
   /* Rellena completamente el vector m */
   llena_vector (m, MAX);
   /* Muestra el vector m antes de ordenarlo */
   printf ("\n\nVector original (Antes de ordenar): \n\n");
   pinta_vector (m, MAX);
   /* Ordena el vector m */
   ordena_vector (m, MAX);
   /* Muestra el vector m despues de ordenarlo */
   printf ("\n\nVector final (Despues de ordenar): \n\n");
   pinta_vector (m, MAX);
   printf ("\n\n\n");
   return (0);
}
```

Ejemplo:version2-makefile.v2

```
/*************************************************************/
// Fichero: funcsvec.h
// Contiene los prototipos de las funciones usadas por la funcion
// "main()" del proyecto de la version 2, asi como la definicion de
// la constante MAX.
/*************************************************************/
#ifndef FUNCSVEC
#define FUNCSVEC

#define MAX          25 // Tamanio del vector

void llena_vector  (int *p, int tope);
void pinta_vector  (int *p, int tope);
void ordena_vector (int *p, int tope);

#endif
```

Ejemplo:version2-makefile.v2

```
/*************************************************************/
// Fichero: funcsvec.c
// Contiene la definicion de las funciones usadas por la
//    funcion "main()" del proyecto de la version 2, ademas de
//    las constantes locales a este modulo
// MAX_LINE y MY_MAX_RAND.
// Los prototipos estan declarados en el fichero "funcsvec.h"
/*************************************************************/
#include <stdlib.h>  // Para manejar nums. aleatorios
#include <time.h>    // Para fijar la semilla del generador de nums. aleat.
#include "funcsvec.h"

#define MAX_LINE     10 // Numero de valores por linea
#define MY_MAX_RAND 100 // Nums. aleatorios entre 0 y 99
void llena_vector  (int *p, int tope) {…}
void pinta_vector  (int *p, int tope) {…}
void ordena_vector (int *p, int tope) {…}
void swap (int *a, int *b){…}
```

Ejemplo:version2-makefile.v2

```
# Fichero: makefile.v2
# Ejemplo de makefile que genera un ejecutable a partir de dos
# ficheros objeto:
#   1) "ppal.o":codigo objeto del programa principal (de "ppal.c").
#   2) "funcsvec.o": codigo objeto de las funciones
#       auxiliares (de "funcsvec.c").
destinos:ordena2.exe clean
ordena2.exe:ppal.o funcsvec.o
     gcc -o ordena2.exe ppal.o funcsvec.o
ppal.o:ppal.c funcsvec.h
     gcc -c -o ppal.o -I. ppal.c
funcsvec.o:funcsvec.c funcsvec.h
     gcc -c -o funcsvec.o -I. funcsvec.c
clean:
     del *.o
```

ordenacion/v2/makefile.v2.mak

Ejemplo:version2-makefile.v2

- En este caso, el ejecutable *ordena2.exe* se construye a partir de los ficheros objeto *ppal.o* y *funcsvec.o*.
- Para generar los ficheros objeto se incluye al fichero de cabecera *funcsvec.h*, de forma que cualquier modificación se propagará a los ficheros que dependen de él.
- La llamada a *gcc* incluye la opción *-I* con el argumento ”.”, para que *gcc* incluya el directorio actual en la lista de búsqueda de los ficheros de cabecera.

Ejemplo: version3-makefil0.v3

```
# Fichero: makefil.v30
# Genera un ejecutable a partir de tres ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal ("ppal.c").
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector.
#   3) "ordena.o": codigo objeto de la funcion de ordenacion.
destinos: ordena.exe clean
ordena.exe : ppal.o vec_ES.o ordena.o
    gcc -o ordena.exe ppal.o vec_ES.o ordena.o
    rename ordena.exe ordena30.exe
ppal.o : ppal.c vec_ES.h ordena.h
    gcc -c -o ppal.o -I. ppal.c
vec_ES.o : vec_ES.c vec_ES.h
    gcc -c -o vec_ES.o -I. vec_ES.c
ordena.o : ordena.c ordena.h
    gcc -c -o ordena.o -I. ordena.c
clean :
    del ppal.o
    del vec_ES.o
    del ordena.o
```

ordenacion/v3/v30/makefil0.v3

Ejemplo: version3-makefil0.v3

- El programa estructura el código en tres ficheros fuente y en dos ficheros de cabecera:
  - *ppal.c* contiene únicamente la función *main()*.
  - El anterior fichero *funcsvec.c* se ha dividido en dos ficheros fuente:
    - Uno conteniendo las funciones que rellenan y muestran el vector (*vec_ES.c*)
    - El otro conteniendo la función de ordenación (*ordena.c*), pensando en un posible uso más general de esta función.
  - Por coherencia, el anterior fichero *funcsvec.h* se ha dividido en dos ficheros de cabecera:
    - *vec_ES.h*
    - *ordena.h*

ordenacion/v3/v30/makefil0.v3

Ejemplo: makefil1.v3

- El programa utiliza una biblioteca creada previamente con el programa *ar*.

```
# Fichero: makefil1.v3
destinos: ordena.exe
ordena.exe : ppal.o libreria.a
    gcc -o ordena.exe ppal.o libreria.a
    rename ordena.exe ordena31.exe
ppal.o : ppal.c vec_ES.h ordena.h
    gcc -c -o ppal.o -I. ppal.c
vec_ES.o : vec_ES.c vec_ES.h
    gcc -c -o vec_ES.o -I. vec_ES.c
ordena.o : ordena.c ordena.h
    gcc -c -o ordena.o -I. ordena.c
libreria.a: ordena.o vec_ES.o
    ar -rsv libreria.a ordena.o vec_ES.o
```

ordenacion/v3/v31/makefil1.v3

Macros en ficheros makefile

- Una **macro o variable MAKE** es una *cadena* que se expande cuando se llama desde un fichero makefile.
- Las macros permiten crear ficheros makefile genéricos o *plantilla* que se adaptan a diferentes proyectos software.
- Una macro puede representar:
  - Listas de nombres de ficheros
  - Opciones del compilador
  - Programas a ejecutar
  - Directorios donde buscar los ficheros fuente
  - Directorios donde escribir la salida, etc.

Macros en ficheros makefile

- La sintaxis de definición de macros en un fichero makefile es la siguiente:

  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.

Macros en ficheros makefile

- Cada macro debe estar **en una línea separada** en un makefile y se sitúan, normalmente, al principio de éste.
- Si make encuentra más de una definición para el mismo **Nombre** (no es habitual), **la nueva definición reemplaza a la antigua**.
- Para expandir una macro escribiremos **$(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:makefil2.v3

```
# Fichero: makefil2.v3 (Version 2 de "makefile.v3")
# Ejemplo de makefile que genera un ejecutable a partir de tres
# ficheros objeto.
#   1) "ppal.o": codigo objeto del programa principal.
#   2) "vec_ES.o": codigo objeto de las funciones de E/S del vector.
#   3) "ordena.o": codigo objeto de la funcion de ordenacion.
# Novedad: uso de macros (OBJ e INCLUDE).
OBJ = ppal.o vec_ES.o ordena.o
INCLUDE = .
destinos: ordena32.exe clean
ordena32.exe : $(OBJ)
    gcc -o ordena32.exe $(OBJ)
ppal.o : ppal.c vec_ES.h ordena.h
    gcc -c -o ppal.o -I$(INCLUDE) ppal.c
vec_ES.o : vec_ES.c vec_ES.h
    gcc -c -o vec_ES.o -I$(INCLUDE) vec_ES.c
ordena.o : ordena.c ordena.h
    gcc -c -o ordena.o -I$(INCLUDE) ordena.c
clean :
    del ppal.o
    del vec_ES.o
    del ordena.o
```

ordenacion/v3/v32/makefil2.v3

Ejemplo:makefil2.v3

- En el ejemplo se define, al principio de las líneas operativas del fichero, la macro llamada **OBJ** cuyo valor es la cadena "`ppal.o vec_ES.o ordena.o`".
- Cuando make procesa makefil2.v3 sustituirá las apariciones de `$(OBJ)` por el valor de la macro `OBJ`, esto es, la regla:

```
ordena1.exe : $(OBJ)
   gcc -o ordena1.exe $(OBJ)
```

la procesa como:

```
ordena1.exe : ppal.o vec_ES.o ordena.o
   gcc –o ordena1.exe ppal.o vec_ES.o
ordena.o
```

Ejemplo:makefil2.v3

- Se define otra macro llamada **INCLUDE** a la que se asigna la cadena "." (el directorio actual). la regla:

```
ordena.o: ordena.c ordena.h
    gcc -c -o ordena.o -I$(INCLUDE)ordena.c
```

- la procesa como:

```
ordena.o : ordena.c ordena.h
    gcc -c -o ordena.o -I. ordena
```

- Si se cambia el directorio de los ficheros de cabecera, sólo hay que modificar el valor de la macro `INCLUDE` del makefile.

Sustituciones de cadenas en macros

- *make* permite sustituir temporalmente caracteres en una macro previamente definida. La sintaxis de la sustitución en macros es la siguiente:

  $(Nombre:TextoOriginal = TextoNuevo)

- Sustituye en la cadena asociada a **Nombre** todas las apariciones de **TextoOriginal** por **TextoNuevo**. Es importante resaltar que:
  - **No** se permiten espacios en blanco antes o después de los dos puntos.
  - **No** se redefine la macro **Nombre**, se trata de una **sustitución *temporal***, por lo que **Nombre** mantiene el valor dado en su definición.
- Por ejemplo:
  - Dada la macro `FUENTE` = f1.c f2.c f3.c se pueden sustituir temporalmente los caracteres `.c` por `.o` escribiendo `$(FUENTE:.c=.o)`.
  - El valor de la macro `FUENTE` no se modifica, la sustitución es temporal.

Ejemplo:makefil3.v3

```
# Novedad: uso de macros con sustitucion.
OBJ  = vec_ES.o ordena.o
INCLUDE = .
MAIN = ppal.o
EXE  = ordena33.exe
VES  = vec_ES
destinos: $(EXE) clean
$(EXE): $(MAIN) $(OBJ)
    gcc $(MAIN) $(OBJ) -o $(EXE)
$(MAIN): $(MAIN:.o=.c) $(VES).h $(VES:vec_ES=ordena).h
    gcc -c -o $(MAIN) -I$(INCLUDE) $(MAIN:.o=.c)
$(VES).o: $(VES).c $(VES).h
    gcc -c -o $(VES).o -I$(INCLUDE) $(VES).c
$(VES:vec_ES=ordena).o: $(VES:vec_ES=ordena).c $(VES:vec_ES=ordena).h
    gcc -c -o $(VES:vec_ES=ordena).o -I$(INCLUDE) $(VES:vec_ES=ordena).c
clean :
    del ppal.o
    del vec_ES.o
    del ordena.o
```

ordenacion/v3/v33/makefil3.v3

Ejemplo:makefil3.v3

Regla 2.

- No se modifica el valor de las macros `MAIN` y `VES`, que siguen teniendo los valores iniciales.

```
$(MAIN): $(MAIN:.o=.c) $(VES).h
$(VES:vec_ES=ordena).h  gcc -c -o $(MAIN)
-I$(INCLUDE) $(MAIN:.o=.c)
```

- es interpretado como:

```
ppal.o: ppal.c vec_ES.h ordena.h
   gcc -c -o ppal.o -I. ppal.c
```

Ejemplo:makefil3.v3

Regla 4.

```
$(VES:vec_ES=ordena).o: $(VES:vec_ES=ordena).c
    $(VES:vec_ES=ordena).h
    gcc -c -o $(VES:vec_ES=ordena).o -I$(INCLUDE)
    $(VES:vec_ES=ordena).c
```

- es interpretado como:

```
ordena.o: ordena.c ordena.h
    gcc -c -o ordena.o -I. ordena.
```

Debemos indicar que no es habitual escribir unas reglas tan complejas para hacer algo tan simple

Macros predefinidas

- Las macros predefinidas utilizadas habitualmente en ficheros makefile son las que enumeramos a continuación.
  - **$^** Equivale a *todas* las dependencias de la regla, con un espacio entre ellas.
  - **$<** Nombre de la *primera dependencia* de la regla.
  - **$@** Nombre del fichero *destino* de la regla.

Macros en las llamadas a *make*

- Puede especificarse el valor de una macro en la llamada a make en lugar de especificar su valor en el fichero makefile.
- Ahora make no busca el valor de la macro en el fichero makefile ya que éste se le pasa como un parámetro más.
- La sintaxis de la llamada a make con macros es la siguiente:

```
make Nombre[= texto a expandir]
     [opciones...] [destino(s)]]
```

- **Nombre**[=**texto a expandir**] define la macro con el nombre **Nombre** con el valor **texto a expandir**.

ordenacion/v3/v34/makefil4.v3

Ejemplo:makefil41.v3

```
# Fichero: makefil4.v3 (Version 4 de "makefile.v3")
# Novedad: uso de macros (DESTDIR) en la llamada a make.
OBJ = ppal.o vec_ES.o ordena.o
INCLUDE = ./include
# DESTDIR = bin
destinos: ordena34.exe clean
ordena34.exe : $(OBJ)
    md $(DESTDIR)
    gcc -o $(DESTDIR)/ordena34.exe $(OBJ)
ppal.o : ppal.c $(INCLUDE)/vec_ES.h $(INCLUDE)/ordena.h
    gcc -c -o ppal.o -I$(INCLUDE) ppal.c
vec_ES.o : vec_ES.c $(INCLUDE)/vec_ES.h
    gcc -c -o vec_ES.o -I$(INCLUDE) vec_ES.c
ordena.o : ordena.c $(INCLUDE)/ordena.h
    gcc -c -o ordena.o -I$(INCLUDE) ordena.c
clean :
    del ppal.o
    del vec_ES.o
    del ordena.o
```

Ejemplo:makefil41.v3

- La segunda regla utiliza la macro `DESTDIR` (directorio donde guardar el fichero *ordena34.exe*) que no está definida en el fichero makefile. Debe se definida en la llamada a *make*.
- Si ejecutamos:

```
make DESTDIR=v34 -f makefil4.v3
```

- la segunda regla se interpretará como sigue:

```
ordena34.exe : ppal.o vec_ES.o ordena.o
    md v34
    gcc -o v34/ordena34.exe ppal.o vec_ES.o
    ordena.o
```

- Si no se especifica el valor de `DESTDIR`, *make* mostrará un mensaje de error (no puede crear el directorio) y terminará de procesar el fichero *makefile* sin generar el ejecutable.

Reglas implícitas

- Las reglas que generan los ficheros objeto son idénticas, sólo se diferencian en los nombres de los ficheros que manipulan.
- Las reglas implícitas son reglas que *make* interpreta para actualizar destinos sin tener que escribirlas dentro del fichero makefile.
- **Si no se especifica una regla explícita para construir un destino**, se utilizará una regla implícita (que no hay que escribir).
- Existe un **catálogo de reglas implícitas** predefinidas.
  - make –p muestra el catálogo
- *make* elegirá una en función del nombre y extensión de los ficheros.

Reglas implícitas

- Las reglas implícitas utilizan una serie de **macros predefinidas**.
  - **CC**: Programa para compilar programas C
  - **CFLAGS**: Opciones de compilación
  - **CPPFLAGS**:Opciones del preprocesador
  - **LDFLAGS**: Opciones del enlazador (ld.exe)
- Estas macros pueden ser RE-definidas:
  - Dentro del fichero makefile,
  - A través de argumentos pasados a make.
  - Con variables del entorno del sistema operativo.
- Si deseamos que *make* use reglas implícitas:
  1. Escribiremos la regla *sin ninguna orden.*
  2. Es posible no escribir la regla.
- Es posible añadir nuevas dependencias a la regla.
- **Ejemplo**: Existe una regla implícita que dice cómo obtener el fichero objeto (.o) a partir del fichero fuente (.c).

```
$(CC) -c $(CPPFLAGS) $(CFLAGS)
```

Reglas implícitas patrón

- Las reglas implícitas patrón pueden ser utilizadas por el usuario para:
  - Definir nuevas reglas implícitas en un fichero *makefile*.
  - Redefinir y adaptar reglas implícitas que proporciona *make*.
- Una *regla patrón* es una regla donde:
  - El destino:
    - Contiene el carácter "*%*" en alguna parte (**sólo una vez**).
    - Constituye un patrón para emparejar nombres de ficheros.
  - Las dependencias
    - Pueden aparecer.
    - Pueden contener el carácter %, incluso varias veces.

Reglas implícitas patrón

- Una regla patrón **`%.o:%.c`** (ó **`c.o:`**) dice cómo construir cualquier fichero .o a partir del fichero .c correspondiente.
- Por ejemplo:

```
%.o : %.c %.h comun.h
        gcc -c $< -o $@
```

  - Significa que cada fichero .o debe volver a construirse cuando se modifique el .c o el .h correspondiente, o bien comun.h.
    - `$<: Primera dependencia de la regla`
    - `$@: Destino de la regla`
- **IMPORTANTE:** sólo se aplicará cuando:
  - El destino no tiene órdenes que lo construya mediante otra regla distinta
  - Puedan encontrase las dependencias.
- Cuando se puede aplicar más de una regla implícita, sólo se aplicará una de ellas: la elección depende del orden de las reglas.

Ejemplo:makefil7.v3

```
# Fichero: makefil7.v3 (Version 7 de "makefile.v3")
# Novedad: uso de reglas implicitas patron.
# NOTA: la regla implícita para enlazar (generar ejecutable) funciona
# correctamente en proyectos con multiples ficheros .o, solo si uno de ellos
# tiene el mismo nombre que el ejecutable

CC = gcc
CPPFLAGS =
CFLAGS = -I$(INCLUDE)
LDFLAGS =
LOADLIBS =
INCLUDE = ./include
ordena37: ordena37.o vec_ES.o ordena.o
ordena37.o : $(INCLUDE)/vec_ES.h $(INCLUDE)/ordena.h
%.o : %.c $(INCLUDE)/%.h
    gcc -c $(CFLAGS) $< -o $@
```

ordenacion/v3/v37/makefil.v7

Ejemplo:makefil7.v3

- La **primera regla** es una regla implícita para el enlazador (para generar el ejecutable).
  - La regla implícita para enlazar funciona correctamente en proyectos con múltiples ficheros objeto solo si uno de ellos tiene el mismo nombre que el ejecutable.
- La **segunda regla** es una regla implícita para el compilador.
  - Los módulos objeto dependen de forma implícita de los módulos fuente asociados (no es necesario indicar esa dependencia).

ordenacion/v3/v37/makefil.v7

Ejemplo:makefil7.v3

- La **tercera regla** es una regla implícita patrón que indica que cada fichero con extensión .o depende de los respectivos ficheros con extensión .c y .h (éstos últimos se encontrarán en el subdirectorio include).
  - El orden en que aparecen las dependencias en la regla es importante pues $< *sustituye a la primera dependencia.*
  - El objeto de esta regla es especificar cómo se construirán los ficheros objeto con la regla anterior.
- La construcción de los ficheros destino se realiza ejecutando la orden asociada a la regla implícita patrón:

```
gcc -c $(FLAGS) $< -o $@
```

  - **$<** : nombre del fichero dependiente.
  - **$@**: nombre del fichero destino.

Ejemplo:makefil7.v3

- Ordena37.o puede construirse con dos reglas diferentes. La que se aplica es la primera que aparece en el fichero makefile, en este caso:

```
ordena37.o : $(INCLUDE)/vec_ES.h
             $(INCLUDE)/ordena.h
```

Reglas patrón estáticas

- Permite aplicar una misma regla a varios destinos. Son muy parecidas en su funcionamiento a las reglas implícitas patrón.
- **No se consideran implícitas**.
- El formato es el siguiente:

```
destino(s): patrón de destino : patrones de dependencia
                                orden(es)
```

  - ***destinos*** especifica a qué destinos se aplicará la regla.
    - **La regla se aplica únicamente a la lista de destinos**, mientras que en las implícitas se intenta aplicar a todos los que se emparejan con el patrón destino.
    - Los destinos pueden contener caracteres comodín como "*" y "?"
  - El ***patrón de destino*** y los ***patrones de dependencia*** dicen cómo calcular las dependencias para cada destino.
- **Ejemplo:** La regla sólo se aplica a los ficheros f1.c y f2.c. Si existen otros ficheros .c, esta regla no se aplicará pues no se han incluido en *destinos*.

```
OBJETOS = f1.o f2.o
$(OBJETOS): %.o: %.c
        gcc -c $(CFLAGS) $< -o $@
```

Directivas condicionales en makefiles

- Se parecen a las directivas condicionales del preprocesador de C.
- Permiten a *make* dirigir el flujo de procesamiento en un fichero makefile en función de la evaluación de una condición en una directiva condicional.
- Las directivas condicionales para ficheros makefile son las siguientes:
  - **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 ó ifneq.

Directivas condicionales en makefiles

```
# Fichero: makefil8.v3 (Version 8 de "makefile.v3")
# Novedad: uso de directivas condicionales en ficheros makefile
CFLAGS = -I$(INCLUDE)
INCLUDE = ./include
ordena38 : ordena38.o vec_ES.o ordena.o
ifndef DESTDIR
     @echo  Error: Falta especificar opcion DESTDIR=directorio
else
     -md $(DESTDIR)
     @echo Creando $@ a partir de: $^
     gcc $^ -o $(DESTDIR)/$@
     @echo
endif
ordena38.o : ordena38.c $(INCLUDE)/*.h
     @echo Creando $@ a partir de: $^
     gcc -c $(CFLAGS) ordena38.c -o $@
     @echo
%.o : %.c $(INCLUDE)/%.h
     @echo Creando $@ a partir de: $^
     gcc -c $(CFLAGS) $< -o $@
     @echo
```

ordenacion/v3/v38/makefil.v8

Ejemplo final

- Escribir un programa que permita al usuario realizar las siguientes operaciones:
  - Mostrar la fecha y hora por pantalla
  - Sacar por pantalla el contenido de un fichero ASCII.
  - Sacar por pantalla el contenido de un directorio
  - Limpiar la pantalla
  - Salir del programa
- Para ello se hará uso de los comandos del sistema operativo:
  - En Linux: *date*, *cat*, *ls*, *clear*
  - En DOS: *date*, *time*, *type*, *dir*, *cls*.
- Estos comandos serán configurados en tiempo de compilación haciendo uso de directivas del compilador.

Ejemplo final

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _WINDOWS_
   #define Directorio "dir“
   #define Fecha "time"
   #define Limpia "cls"
   #endif
#ifdef _LINUX_
   #define Directorio "ls”
   #define Fecha "date“
   #define Limpia "clear”
#endif

int main(int argc, char ** argv){
   system(Limpia);
   system(Directorio);
   system(Fecha);
   return 0;
}
```

Ejemplo final

- Para compilar:
  - *gcc -D_LINUX_ linuxWin.c*
- Con un fichero makefile:
  - *Make –f linuxWin.mak SISTEMA=_WINDOWS*

```
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
```