



ENTORNO DE DESARROLLO INTEGRADO NETBEANS

Herramientas de depuración

La tarea de depuración de un programa consiste básicamente en explorar dicho programa, ejecutándolo paso a paso y comparando en todo momento los valores que van tomando las distintas variables con los valores esperados.

Es necesario haber diseñado con anterioridad un conjunto de casos de prueba que nos permitan anticipar en todo momento los valores que deben tomar las variables y saber si la progresión del programa es correcta o no.

Normalmente la depuración se realizará ante la presencia de errores de ejecución o lógicos, ya que los de sintaxis suelen ir acompañados de mensajes de error facilitados por el compilador, lo que facilita su localización.

Las herramientas de IDE (Integrated Development Environment) disponen de funcionalidades específicas para ayuda a la depuración de programas. Dichas funcionalidades permiten realizar, entre otras, las siguientes tareas:

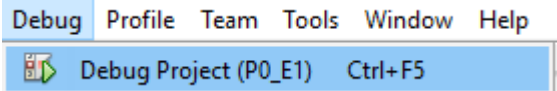

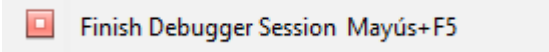

- Ejecución paso a paso del programa
- Evaluación de los valores de las variables, arrays, ...
- Establecimiento de puntos de detención temporal del flujo del programa

En primer lugar, y para tener un ejemplo básico sobre el que probar las herramientas de depuración, crea un proyecto **P0_E1** (abre NetBeans y selecciona *File->New Project*. A continuación selecciona *Categoría: C/C++* y *Proyectos: C/C++ Application* y pulsa *Next*>. En la siguiente pantalla debes seleccionar el nombre del proyecto (**P0_E1**) y su localización (aquí puedes poner la carpeta donde vas a guardar las prácticas de esta asignatura, tu home de cygwin `C:\cygwin64\home\usuario` o una unidad de pendrive, estas dos últimas opciones son las más cómodas para trabajar con makefiles desde la ventana de comandos de cygwin). Marca la opción "Create Main File" y fíjate que a su derecha esté seleccionado "C/C11", y no "C++". Al pulsar "Finish" se creará la carpeta **P0_E1** en tu carpeta de usuario y escribirá un prototipo de `main.c` (lo puedes ver si pulsas el + a la izquierda de *Source Files* en la pestaña de Proyectos de NetBeans) que debes modificar para que contenga lo siguiente a partir de la línea 20:


```
20  int main(int argc, char** argv) {
21
22      int a = -1, b = -1, c = -1;
23      printf("Buenas tardes. Digame dos numeros: ");
24      scanf("%d %d", &a, &b);
25      c = a + b;
26      printf("\n%d+%d=%d\n", a, b, c);
27
28      return (EXIT_SUCCESS);
29 }
```

PRÁCTICA 0: DEPURACIÓN Y CONSTRUCCIÓN DE MAKEFILE

Activación del modo depuración¹

Barra de menús	Debug→Debug Project	
Barra de herramientas	Botón "Debug Project"	
Atajo de teclado	[CTRL+F5]	
Abandono del modo depuración		
Barra de menús	Debug→Finish Debugger Session	
Barra de herramientas	Botón "Finish Debugger Session"	
Atajo de teclado	[MAY+F5]	

Puntos de interrupción (breakpoint)

La tarea básica de depuración de un programa suele ser la ejecución paso a paso del mismo, comenzando en aquella sentencia donde se sospecha que se encuentra el error. Para detener la ejecución del programa en una sentencia se utilizan los puntos de interrupción (*breakpoints*). Para fijar un punto de interrupción en una línea basta con hacer clic en el margen izquierdo de dicha línea. El IDE visualiza el punto de detención mediante un símbolo  en el lugar del número de la línea y un sombreado de color rojo en la línea correspondiente:

```

20  □ int main(int argc, char** argv) {
21
22      int a = -1, b = -1, c = -1;
23      printf("Buenas tardes. Digame dos numeros: ");
24      scanf("%d %d", &a, &b);
25  □  c = a + b;
26      printf("\n%d+%d=%d\n", a, b, c);
27
28      return (EXIT_SUCCESS);
29  }
```

PUNTO DE INTERRUPCIÓN EN LA LÍNEA 25

Los puntos de interrupción se eliminan de la misma forma que se establecen: haciendo clic en el margen de la línea del programa.

Ejecuta este programa **EN MODO DEPURACIÓN** y observa que, después de introducir los dos números solicitados, el flujo se detiene en el punto de interrupción¹. Esta situación se identifica mediante un símbolo de flecha y un sombreado verde en la línea correspondiente:

¹ Si al ejecutar el modo de depuración sale una ventana de "Warning: Selected console type is not supported in your configuration, using External terminal instead" aceptamos y el programa se ejecutará en una ventana de comandos emergente. Si no sale esta ventana de warning, probablemente sea porque necesita ejecutarse en un terminal externo y está puesto "Internar Terminal" por defecto. Haz clic con el botón derecho del ratón sobre el nombre del proyecto y selecciona "Properties". En la Categoría "Run" elige, en "Console Type", la opción "Standard output" (que hará que salga el aviso) o "External Terminal" y pulsa el botón OK.

PRÁCTICA 0: DEPURACIÓN Y CONSTRUCCIÓN DE MAKEFILE

```
20 int main(int argc, char** argv) {
21
22     int a = -1, b = -1, c = -1;
23     printf("Buenas tardes. Digame dos numeros: ");
24     scanf("%d %d", &a, &b);
25     c = a + b;
26     printf("\n%d+%d=%d\n", a, b, c);
27
28     return (EXIT_SUCCESS);
29 }
```








DETENCIÓN DEL FLUJO EN EL PUNTO DE INTERRUPCIÓN DE LA LÍNEA 25

La línea verde indica en todo momento el punto donde se ha detenido el flujo de ejecución del programa. Hay que tener en cuenta que la sentencia donde se ha detenido el programa **NO SE HA EJECUTADO TODAVÍA**, sino que será la siguiente en ejecutarse.

Ejecución paso a paso

El flujo de ejecución se realiza con normalidad hasta el punto de interrupción. A partir de ahí, con el programa detenido, es posible reanudar la ejecución de varias formas. La más habitual es la ejecución paso a paso, que nos permite reanudar el flujo ejecutando cada sentencia individualmente. La ejecución paso a paso se puede realizar de varias formas:

Ejecución paso a paso

Barra de menús	Debug→Step into	 Step Over F8  Step Over Expression Mayús+F8  Step Into F7 Step Into Next Method Mayús+F7  Step Out Ctrl+F7  Run to Cursor F4  Step Into Last Function
Barra de herramientas	Botón "Step into"	
Atajo de teclado	[F7]	

Con cada pulsación de F7 el flujo avanza una sentencia. Esta situación se visualiza en el IDE con el avance de la línea verde que indica el lugar donde se ha detenido el flujo del programa.

PRÁCTICA 0: DEPURACIÓN Y CONSTRUCCIÓN DE MAKEFILE

```
20  □ int main(int argc, char** argv) {
21
22      int a = -1, b = -1, c = -1;
23      printf("Buenas tardes. Digame dos numeros: ");
24      scanf("%d %d", &a, &b);
25      □ c = a + b;
26      printf("\n%d+%d=%d\n", a, b, c);
27
28      → return (EXIT_SUCCESS);
29  }
```

DETENCIÓN DEL FLUJO DEL EJEMPLO ANTERIOR TRAS LA EJECUCIÓN PASO A PASO DE DOS SENTENCIAS

Hay que tener en cuenta que si a medida que avanza el flujo del programa se ejecutan sentencias de E/S, el cuadro de diálogo correspondiente se establecerá mediante el mecanismo habitual (típicamente otra ventana). En el ejemplo anterior, la ejecución de las sentencias 23 y 24 provoca la apertura de una ventana donde el usuario interactúa con el programa, de la forma habitual:

```
Buenas tardes. Digame dos números: 2 3
```

Ejercicio 1

Sobre el programa ejemplo, incluye un punto de interrupción en la primera línea ejecutable (la línea 22 donde declaras las variables) y realiza la ejecución paso a paso del programa, desde la primera línea a la última (lo más cómodo es pulsar sucesivamente la tecla F7).

Visualización de valores de variables

La ejecución paso a paso de un programa es una herramienta útil en la depuración cuando se combina con la visualización del valor de las diferentes variables declaradas en el mismo. Esto permite ir siguiendo las diferentes operaciones (aritméticas, lógicas, asignaciones) del programa, lo que facilita la detección de los errores.

La herramienta de visualización de variables es accesible desde NetBeans en cualquier momento durante la ejecución paso a paso. Para acceder a ella basta activar la solapa "Variables". Debido a la configuración y actualización de NetBeans, puede que estas pestañas cambien de sitio, pero siempre las encontraréis en el menú **Window→Debugging**.

Output	Variables ×	Call Stack	Breakpoints	Sessions
	Name		Value	
	<Enter new watch>			
		argc	1	
		argv	0xffffcc10	
		a	0	
		b	0	
		c	0	

En el ejemplo anterior, vemos el estado de las variables del programa antes de la ejecución de la primera sentencia:

PRÁCTICA 0: DEPURACIÓN Y CONSTRUCCIÓN DE MAKEFILE

```
20 int main(int argc, char** argv) {
21
22     int a = -1, b = -1, c = -1;
23     printf("Buenas tardes. Digame dos numeros: ");
24     scanf("%d %d", &a, &b);
25     c = a + b;
26     printf("\n%d+%d=%d\n", a, b, c);
27
28     return (EXIT_SUCCESS);
29 }
```

También es posible **evaluar expresiones** en lugar de variables. NetBeans denomina a estas expresiones "elementos observados" o "watches". Se puede crear un elemento observado mediante:

Creación de watches

El botón "Create a new watch" de la solapa "Variables"



El menú Debug→New watch...

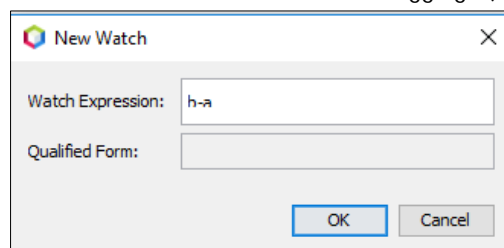
New Watch...

Ctrl+Mayús+F7

El atajo de teclado

[CTRL+MAYÚS+F7]

En la ventana que surge puede escribirse la expresión a evaluar, a la que podrá accederse desde la ventana "Variables" o "Watches" (si no aparece, se puede activar con el menú Window→Debugging...)



En la pestaña "Variables" es posible visualizar conjuntamente variables y elementos observados pulsando el primer botón, en otro caso pueden verse separados en las pestañas "Watches" y "Variables":

Output	Watches	Variables ×	Call Stack	Breakpoints	Sessions
	Name	Value			
	b-a	0			
	a+b	0			
	b-2*a	0			
	<Enter new watch>				
	argc	1			
	argv	0xffffcc10			
	a	0			
	b	0			
	c	0			

PRÁCTICA 0: DEPURACIÓN Y CONSTRUCCIÓN DE MAKEFILE

Ejercicio 2


Sobre el ejemplo anterior, ejecuta el programa visualizando el valor de cada variable. Completa la tabla siguiente, anotando los valores de las diferentes variables ANTES de la ejecución de las líneas respectivas:

LÍNEA	a	b	c
22			
23			
25			
26			

Justifica en todo momento los resultados en función de la sentencia que se está ejecutando en cada instante.

Ejercicio 3 (versión 1) (la versión entregable se realizará con makefile)

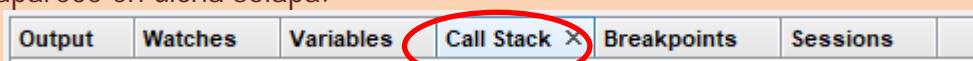
Crea el proyecto **P0_E3** desde NetBeans en tu carpeta de trabajo (sin crear main file), descarga el archivo **P0_E3.zip** del campus virtual y guarda los ficheros que contiene (**main.c**, **potencia.c** y **factorial.c**) en la carpeta **P0_E3** que creó NetBeans para el proyecto. Desde NetBeans, haz clic sobre el símbolo "+" a la izquierda del proyecto y con el botón derecho sobre "Source Files". Elige la opción "Add Existing Item..." y añade, dentro de la carpeta **P0_E3**, los archivos **.c** que acabas de copiar.

- Compila el proyecto **P0_E3** y realiza la ejecución paso a paso. Encuentra los errores lógicos que hay en el programa. ¿Qué instrucciones están mal y por qué instrucciones las has sustituido? (*no borres las instrucciones incorrectas, coméntalas y escribe las nuevas*). Comprueba que el programa funciona correctamente para todos los casos que se indican en los comentarios de **main.c**.
- Observa qué sucede al ejecutar paso a paso (botón "Step into" [F7]) la sentencia donde se invoca la función **factorial()** y continúa la ejecución a partir de ahí.
- Al llegar a la función **potencia()**, utiliza el botón "STEP OVER" [F8]  en lugar del botón "Step into" [F7] ¿Qué diferencias observas frente a la ejecución del apartado b?

Ejercicio 4

Repite el ejercicio 3 visualizando todas las variables del programa. Indica qué sucede con la ventana de variables cuando el flujo del programa continúa dentro de la función.

Observa en ese momento la solapa "pila de llamadas" o "Call Stack" y justifica la información que aparece en dicha solapa.



Ejercicio 5 (versión 1) (la versión entregable se realizará con makefile)

Implementa un programa para elevar a la potencia **b** cada uno de los elementos de un array **a**: $c=a^b$, siendo a y b las variables siguientes: `int a[4], int b;` de modo que el vector potencia **c** se calcule mediante una función `potencia()`. Puedes usar como base la función potencia utilizada en el ejercicio 3.

Ejecuta paso a paso el programa, visualizando en todo momento las variables, y observando cómo se modifican las componentes de los arrays.

Utilizando el botón "Watches", visualiza las siguientes expresiones, que indican la dirección de memoria de los respectivos elementos del array: `&a[0] &a[1] &a[2] &a[3]`.

¿Cuál es el mayor y cuál el menor? Calcula la diferencia entre las direcciones de dos elementos consecutivos. Trata de justificar los resultados.

Ejercicio 6 (versión 1) (la versión entregable se realizará con makefile)

Crea el proyecto **P0_E6** desde NetBeans en tu carpeta de trabajo (sin crear main file), descarga el archivo **P0_E6.zip** del campus virtual y guarda los ficheros que contiene en la carpeta **P0_E6** que creó NetBeans para el proyecto. Desde NetBeans, haz clic sobre el símbolo "+" a la izquierda del proyecto y con el botón derecho sobre "Header Files". Elige la opción "Add Existing Item..." y añade, dentro de la carpeta **P0_E6**, el archivo `matriz.h` que acabas de copiar. Repite la operación sobre "Source Files", elige la opción "Add Existing Item..." y añade, dentro de la carpeta **P0_E6**, los archivos .c que acabas de copiar. Estos ficheros contienen la implementación de una matriz de datos y las operaciones básicas asociadas.

El proyecto contiene algunos errores lógicos, que deberás detectar usando las técnicas indicadas en este guion.

Corrige los errores y prueba el programa con los casos de uso que se indican en los comentarios iniciales de `main.c`.

¿Se está liberando correctamente la memoria en todos los casos? Una vez modificada la liberación de memoria, prueba a crear las matrices, destruirlas y salir del programa (opción '0'). ¿Está finalizando la ejecución de forma correcta?

COMPILACIÓN DESDE LÍNEA DE COMANDOS

Partiendo del ejercicio 6 corregido, vamos a realizar la compilación y ejecución del programa desde línea de comandos. Si estás trabajando en Windows, lo más práctico es que copiéis la carpeta del proyecto en la carpeta home de cygwin (C:/cygwin64/home/nombreusuario) o en vuestra unidad de pendrive, para poder realizar la compilación por línea de comandos de forma cómoda. Toda esta parte de la práctica se trabajará con la ventana de comandos y el explorador de archivos. Vamos a poner un ejemplo en el que los archivos .h estén en otra carpeta, pero no tienen por qué estar en un sitio distinto a los demás archivos del proyecto. **El objetivo es que tengáis un makefile base a partir del cual podáis compilar cualquier proyecto con pequeñas adaptaciones.**

1. En primer lugar, copia todos los ficheros fuente (.c) del ejercicio 6 corregido en una nueva carpeta **P0_E6_make**. Crea dentro de esa carpeta otra carpeta llamada **headerFiles** y mueve a ella el fichero de cabecera (.h) **matriz.h**. Esto lo puedes hacer desde la ventana de comandos con los comandos **mkdir** y **mv**.
2. La creación de un ejecutable desde línea de comandos consta de dos pasos. En primer lugar es necesario transformar los ficheros .c (archivos fuente) en ficheros .o (archivos objeto) indicando con la opción **-I** en qué lugar se encuentran los archivos de cabecera o .h:

```
$ gcc -c asignarElemento.c destruirMatriz.c main.c suma.c crearMatriz.c  
imprimirMatriz.c obtenerElemento.c producto.c -I ./headerFiles
```

3. A continuación, para generar el fichero ejecutable debemos enlazar todos los ficheros .o:

```
$ gcc -o ejecutable asignarElemento.o destruirMatriz.o main.o suma.o  
crearMatriz.o imprimirMatriz.o obtenerElemento.o producto.o
```

4. Para ejecutar el programa:

```
$ ./ejecutable
```

5. Por último, borra los archivos .o:

```
$ rm *.o
```

CONSTRUCCIÓN DE MAKEFILE

Partiendo del paso anterior (compilación y ejecución desde línea de comandos), vamos a crear un makefile. Makefile es una herramienta de Unix/Linux que permite simplificar la compilación y la creación de ficheros ejecutables. Utilizar el método descrito en el paso anterior presenta dos inconvenientes:

- El primero es tener que reescribir el comando cada vez que queremos compilar (si usamos el mismo terminal podemos recurrir a la historia de comandos para evitar esto).
- En segundo lugar, aun cuando solo cambie un fichero, el comando compilará de nuevo todos los archivos.

El uso de makefile evita todos estos inconvenientes.

A continuación tienes los pasos para crear un fichero **makefile**, que siempre se escribe con al menos un objetivo, unas dependencias y unas instrucciones en la forma:

objetivo: dependencias (todo lo que se necesita para poder ejecutar las instrucciones)
instrucciones

1. Crea un fichero con el nombre **makefile** en el mismo directorio que los ficheros fuente con el contenido:

Regla: **prerrequisitos** → **OUTPUT:** `asignarElemento.c destruirMatriz.c main.c suma.c crearMatriz.c
imprimirMatriz.c obtenerElemento.c producto.c ./headerFiles/matriz.h`
Tabulador → `gcc -Wall -o ejecutable asignarElemento.c destruirMatriz.c main.c
suma.c crearMatriz.c imprimirMatriz.c obtenerElemento.c producto.c -I
./headerFiles`

Las reglas en **make** se identifican por "**nombreRegla**:". A la izquierda de ":" está el nombre de la regla y a la derecha sus dependencias (**la regla se ejecuta si alguna de sus dependencias ha cambiado**). El comando

PRÁCTICA 0: DEPURACIÓN Y CONSTRUCCIÓN DE MAKEFILE

asociado a la regla aparece en la siguiente línea, y debe incluir un **tabulador al comienzo de la línea** (si no, no funcionará). Como ejemplo, en este caso la regla se llama **OUTPUT** y como dependencias tiene todos los ficheros `.c` y `.h`, es decir, si alguno de ellos es modificado, la regla se ejecuta completamente.

2. Teclea desde línea de comandos **make**.

Como no tiene argumentos, buscará un fichero denominado `makefile` y ejecutará la primera regla de dicho fichero. Si el `makefile` estuviese en otro fichero, por ejemplo `makefilev0`, el comando a ejecutar sería `"make -f makefilev0"`.

Prueba a cambiar el nombre del fichero `makefile` por `makefilev0` mediante el comando `mv` (`$ mv makefile makefilev0`) y comprueba que compila.

Vamos a mejorar el `makefile` un poco más creando un conjunto de variables.

Copia `makefilev0` en `makefilev1` (usando el comando `cp`: `$cp makefilev0 makefilev1`) y edita este último para modificarlo del siguiente modo:

```
#opciones de compilación, muestra todos los warnings (-Wall)
CC=gcc -Wall
#si incluye una librería estándar, en este caso la matemática (fichero libm.a)
#todas tienen el formato de fichero libNOMBRE.a
#y al incluirla en el compilador se pone -lNOMBRE
#si no hay librerías adicionales, se elimina esta línea
LIBS = -lm

#carpeta de las cabeceras (si están en la actual, ponemos .)
HEADER_FILES_DIR = ./headerFiles
#opciones de compilación, indica dónde están los archivos .h
INCLUDES = -I $(HEADER_FILES_DIR)

#nombre del ejecutable o archivo de salida
OUTPUT = ejecutable

#ficheros .h. Si hay varios, se enumeran con su path completo
LIB_HEADERS = $(HEADER_FILES_DIR)/matriz.h

#fuentes
SRCS = asignarElemento.c destruirMatriz.c main.c suma.c crearMatriz.c
imprimirMatriz.c obtenerElemento.c producto.c

#regla (dependencia de los .c y los .h)
#si no hay librerías adicionales, no existe la variable $(LIBS),
#por lo que se elimina $(LIBS) de la regla siguiente
$(OUTPUT): $(SRCS) $(LIB_HEADERS)
    $(CC) -o $(OUTPUT) $(SRCS) $(INCLUDES) $(LIBS)
```

Las variables se definen por medio de "=", y se referencian usando `$(NOMBRE_VARIABLE)`. La librería matemática (opción `-lm`) no es necesaria para el ejercicio 6, pero se ha añadido aquí para que veáis como incluir una librería.

Ejecuta el comando `make` con esta versión del `makefile`: `$make -f makefilev1`

Como no hemos modificado ningún fichero nos dirá que "ejecutable está actualizado". Eso es porque hemos puesto como dependencias de la regla los ficheros `.c` y `.h` y, como estos no los hemos cambiado desde la última compilación, no tiene nada que hacer.

Borra el fichero `ejecutable` (`$rm ejecutable.exe`) y vuelve a ejecutar el comando `make`: `$make -f makefilev1`.

PRÁCTICA 0: DEPURACIÓN Y CONSTRUCCIÓN DE MAKEFILE

Comprueba que, con este **makefile**, al modificar un fichero fuente (.c) o un fichero .h se recompilan TODOS LOS ARCHIVOS.

Para que esto no suceda, sino que sólo se recompilen los archivos modificados, vamos a cambiar el **makefile** para que se generen los ficheros con código objeto (.o) para cada archivo fuente (.c), que son los ficheros compilados pero sin enlazar. Estos archivos objeto se generan con la opción **-c** de **gcc**. Para enlazar todos los ficheros objeto, se utiliza la opción **gcc** pero incluyendo los ficheros .o en lugar de los ficheros .c, y por lo tanto estos archivos .o serán dependencias para poder ejecutar la regla de compilación, como podemos ver en la siguiente modificación. Siempre se ejecuta la primera regla, pero si esta depende de otras reglas, se resuelven en primer lugar las dependencias entre ellas. Copia **makefilev1** en **makefilev2** usando el comando **cp (\$cp makefilev1 makefilev2)** y edita este último para modificarlo como sigue:

```
#opciones de compilación, muestra todos los warnings (-Wall)
CC=gcc -Wall
#si incluye una librería estándar, en este caso la matemática (fichero libm.a)
#todas tienen el formato de fichero libNOMBRE.a
#y al incluirla en el compilador se pone -lNOMBRE
#si no hay librerías adicionales, se elimina esta línea
LIBS = -lm

#carpeta de las cabeceras (SI ESTÁN EN LA ACTUAL, PONEMOS .)
HEADER_FILES_DIR = ./headerFiles
#opciones de compilación, indica dónde están los archivos .h
INCLUDES = -I $(HEADER_FILES_DIR)

#nombre del ejecutable o archivo de salida
OUTPUT = ejecutable

#ficheros .h. Si hay varios, se enumeran con su path completo
LIB_HEADERS = $(HEADER_FILES_DIR)/matriz.h

#fuentes
SRCS = asignarElemento.c destruirMatriz.c main.c suma.c crearMatriz.c
imprimirMatriz.c obtenerElemento.c producto.c

#ficheros .o: todos los .o con un análogo .c en SRCS
OBJS = $(SRCS:.c=.o)

#regla 1 (dependencia de los .o)
#si no hay librerías adicionales, no existe la variable $(LIBS),
#por lo que se elimina $(LIBS) de la regla siguiente
$(OUTPUT): $(OBJS)
    $(CC) -o $(OUTPUT) $(OBJS) $(LIBS)

#regla 2 (genera los .o cuando es necesario, dependencia de los .c y .h)
#$$ es el nombre del fichero que se genera con la regla (.o)
#$$ es el nombre del primer prerrequisito (el archivo .c cuyo .o se está
generando)
%.o: %.c $(LIB_HEADERS)
    $(CC) -c -o $$ $(INCLUDES)
```

La variable **OBJS** tomará los mismos valores que la variable **SRCS** pero cambiando ".c" por ".o".

PRÁCTICA 0: DEPURACIÓN Y CONSTRUCCIÓN DE MAKEFILE

La primera regla del make es similar a la anterior, pero incluyendo los ficheros `.o` (**OBJS**), por tanto no es necesario indicar la carpeta donde están los ficheros `.h` (**INCLUDES**) (esto hace falta al generar los `.o`, por eso está en la segunda regla, y es coherente con cómo hicimos la compilación desde ventana de comandos al principio del ejercicio (página 8)).

La segunda regla del `makefile` permite generar todos los ficheros `".o"` sin necesidad de enumerarlos uno a uno. Además, establece la dependencia de cada `".o"` con su respectivo `".c"` y también con los ficheros definidos en `LIB_HEADERS` (en este caso el fichero `matriz.h`).

Ejecuta el comando make con esta versión del makefile: `$make -f makefilev2`

Fíjate cómo se va ejecutando la segunda regla para generar todos los archivos `.o` y, una vez creados todos los ficheros necesarios para ejecutar la primera regla, se ejecuta esta, creando el fichero ejecutable.

Ahora, modifica un fichero `".c"` y vuelve a lanzar make. Fíjate que sólo se compila el fichero modificado y se crea el ejecutable.

¿Qué ocurre si se modifica el fichero `matriz.h`? ¿Por qué?

EN MAKEFILE, LAS REGLAS SE EJECUTAN POR ORDEN DE APARICIÓN. Si queremos ejecutar la regla 2, o la 3 o la 4 de forma individual y no por orden, tendremos que invocarla directamente: `make REGLA2` (en nuestro caso `make -f makefilev2 REGLA2`). Fíjate que la regla 2 en este caso representa múltiples reglas, todas las que generan los ficheros `.o` necesarios (como has visto en la ejecución), por lo que si quieres generar uno de los ficheros `.o` de forma individual podrías hacerlo invocando esa regla concreta: `make -f makefilev2 crearMatriz.o`.

Para finalizar, vamos a completar el `makefile` con un par de nuevas reglas.

Copia `makefilev2` en `makefilev3` usando el comando `cp` (`$cp makefilev2 makefilev3`) y edita este último para añadirle al final las reglas que se indican a continuación:

```
# regla 3 que borra el ejecutable (prerrequisito: clean)
cleanall: clean
    rm -f $(OUTPUT)
#regla 4 que borra los ficheros .o y los de backup (terminan en ~)
clean:
    rm -f *.o *~
```

Ejecuta `make` con esta versión de makefile: `$make -f makefilev3`

A continuación, ejecuta `"make clean"` (es necesario poner la regla que se quiere ejecutar si ésta no es la primera del `makefile`). Como tu `makefile` no se llama así, sino `makefilev3`, tendrás que escribir `$make -f makefilev3 clean` ¿Qué sucede?

Repite el proceso para la otra regla `"make cleanall"`: `$make -f makefilev3 cleanall` ¿Qué sucede?

Por último, lo habitual es que todas las funciones enumeradas en `matriz.h` **NO** estén en ficheros independientes, como es este caso, sino que estén en un único archivo que tenga el mismo nombre que el `.h`, pero con `.c`, es decir, en un único archivo `matriz.c`. En ese caso, también se puede poner una regla para tener esto en cuenta en el `makefile`.

Crea, dentro de la carpeta `headerFiles`, el archivo `matriz.c` y copia dentro de él el código de todos los ficheros fuente: `crearMatriz.c`, `destruirMatriz.c`, `asignarElemento.c`, `imprimirMatriz.c`, `obtenerElemento.c`, `suma.c` y `producto.c`

Copia `makefilev3` en `makefilev4` usando el comando `cp` (`$cp makefilev3 makefilev4`) y edita este último para **modificar** la definición de la variable `SRCS` y **añadir** a la regla `clean` la eliminación de los ficheros `.o` que estén dentro de la carpeta `headerFiles`, es decir, `matriz.o`, como se indica a continuación:

...

PRÁCTICA 0: DEPURACIÓN Y CONSTRUCCIÓN DE MAKEFILE

```
#fuentes
SRCS = main.c $(LIB_HEADERS:.h=.c)
...
#regla 4 que borra los ficheros .o y los de backup (terminan en ~)
clean:
    rm -f *.o $(HEADER_FILES_DIR)/*.o *~
```

Ejecuta **make** con esta versión de makefile: `$make -f makefilev4`

A continuación, ejecuta "**make clean**" (es necesario poner la regla que se quiere ejecutar si ésta no es la primera del makefile). Como tu makefile no se llama así, sino **makefilev4**, tendrás que escribir `$make -f makefilev4 clean ¿Qué sucede?`

Puedes usar como base **makefilev3** para proyectos en los que el fichero **.h** NO tiene su equivalente **.c** y **makefilev4** para los proyectos en los que los ficheros **.h** SÍ tienen su equivalente **.c**, que será lo habitual. En el caso de los ejercicios entregables de esta práctica, usa **makefilev3** renombrándolo como **makefile** por simplicidad y adaptándolo a cada caso (p.e. en los ejercicios 3 y 5, no hay ficheros **.h** ni hay librerías adicionales).

Para ampliar la información relacionada con la generación de makefiles puedes consultar las siguientes URLs:

- <http://mrbook.org/blog/tutorials/make/>
- <http://arco.esi.uclm.es/~david.villa/doc/repo/make/make.html>

ENTREGABLES

Deberás entregar a través del Campus Virtual los ejercicios 3 (factorial y potencia), 5 (potencia de array) y 6 (matrices) construyendo su propio makefile. Para ello puedes usar como base **makefilev3** y eliminar todo lo que no se use en tu proyecto (p.e. si no hay archivos **.h** o no se añaden librerías adicionales). Las fechas de entrega se especifican en el Campus Virtual, y las instrucciones para generar el fichero son las siguientes:

- Deberás subir un único fichero comprimido zip con el nombre **apellido1_apellido2.zip**.
- El fichero comprimido contendrá una carpeta por cada ejercicio.
- De cada ejercicio incluirás **ÚNICAMENTE** los ficheros fuente (.c) y de cabecera (.h), así como su correspondiente makefile.

Todos los ejercicios deberán compilar directamente con el makefile (sin opciones) en Cygwin/Linux, en otro caso serán evaluados con la calificación de 0. Este criterio se mantendrá en el resto de prácticas de la asignatura.