UNIVERSIDAD DE GRANADA

Guía Práctica de Compilación en C++

Metodologías de la Programación

Grado en Ingeniería Informática

David A. Pelta

El contenido de esta guía es una adaptación del material desarrollado por el Dr. Eduardo Eisman para la asignatura Metodologías de la Programación II, de la Ing. Superior en Informática para el Curso 2008-2009

Antes de empezar a crear programas y compilarlos, vamos a establecer una estructura de directorios que mantendremos a lo largo de toda la asignatura. De esta forma tendremos perfectamente organizados todos nuestros ficheros con código fuente, ficheros de cabecera, ejecutables, etc. Lo primero que vamos a hacer es ***crear un directorio*** para la asignatura, que vamos a llamar **mp**. Para crearlo, abrimos una nueva consola (una terminal) y tecleamos ***mkdir mp***.

```
profesor@DECSAI:~$ mkdir mp
profesor@DECSAI:~$
```

Es importante aclarar que, al contrario de lo que ocurre en Windows, Linux es ***sensible a mayúsculas***, es decir, que podemos crear por ejemplo los directorios **mp**, **Mp** y **MP**, y los tres serán directorios diferentes. Para ver que efectivamente se ha creado un directorio llamado **mp**, podemos ***mostrar el contenido del directorio*** actual ejecutando la orden ***ls***.

```
profesor@DECSAI:~$ ls
Documentos  Escritorio  Examples  Imágenes  mp  Música  Plantillas  Público  Vídeos
profesor@DECSAI:~$
```

En cualquier momento podemos ***hacer uso de la ayuda*** con la orden ***man*** (podemos saber para qué sirve un comando, ver qué argumentos acepta, etc.). Por ejemplo, si escribimos ***man ls***, podemos ver que, con el argumento **-a**, ***ls*** muestra ***todo el contenido*** del directorio (incluidos los ficheros que oculta por defecto, los cuales comienzan por un punto), o que con **-l** la información que se muestra es mucho ***más detallada*** (permisos que se tienen sobre el archivo, tamaño, etc.). Para navegar por la ayuda podemos usar las flechas del teclado o las teclas de avanzar y retroceder página, y para ***salir*** sólo tenemos que pulsar la tecla **q**.

```
profesor@DECSAI:~$ man ls

LS(1)                                                User Commands                                                LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by default).  Sort entries alphabetically if none of -cftuvSUX nor --sort.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .
q
profesor@DECSAI:~$ profesor@DECSAI:~$
```

Para ***entrar en el directorio*** que acabamos de crear escribimos ***cd mp***.

```
profesor@DECSAI:~$ cd mp
profesor@DECSAI:~/mp$
```

Si volvemos a listar el contenido con ***ls***, veremos que el directorio **mp** no contiene nada, está vacío.

```
profesor@DECSAI:~/mp$ ls
profesor@DECSAI:~/mp$
```

Ahora vamos a crear una serie de directorios para mantener todos nuestros ***ficheros organizados***: **bin** (aquí guardaremos los ***ejecutables*** que vayamos creando, que tendrían una extensión **.exe** en Windows, aunque en Linux los ejecutables no tienen extensión), **doc** (para la ***documentación***), **include** (para los ***ficheros de cabecera***, es decir, los **.h**), **lib** (para las ***bibliotecas***, cuya extensión es **.a**), **obj** (para los ***ficheros objeto***, **.o**) y **src** (para el ***código fuente***, es decir, los archivos con extensión **.cpp**). Todos estos directorios los vamos a crear con la orden ***mkdir nombreDelDirectorio***.

mp
- **bin**
- **doc**
- **include**
- **lib**
- **obj**
- **src**

```
profesor@DECSAI:~/mp$ mkdir bin
profesor@DECSAI:~/mp$ mkdir doc
profesor@DECSAI:~/mp$ mkdir include
profesor@DECSAI:~/mp$ mkdir lib
profesor@DECSAI:~/mp$ mkdir obj
profesor@DECSAI:~/mp$ mkdir src
profesor@DECSAI:~/mp$
```

Ahora que ya tenemos creados nuestros directorios, podemos empezar a trabajar. Lo primero que vamos a hacer es un programa muy sencillo para ver cómo se crea desde la terminal un nuevo fichero en el que introduciremos el código fuente, y cómo se compila este. El programa es el típico ***"Hola mundo"*** (***Hello world***) que siempre se crea cuando se aprende un nuevo lenguaje de programación. Vamos a crear un fichero de texto llamado **HolaMundo.cpp**. Para ello, tras asegurarnos que estamos en el directorio **mp** que hemos creado anteriormente, escribimos en la consola ***kwrite src/HolaMundo.cpp***.

```
profesor@DECSAI:~/mp$ kwrite src/HolaMundo.cpp
```

De esta forma, con el programa ***kwrite*** (uno de los muchos editores de texto existentes en Linux junto con ***kate***, ***gedit***, ***vi***…) crearemos un nuevo fichero llamado **HolaMundo.cpp** en el directorio **src** que cuelga del directorio **mp** en el que nos encontramos. Ahora tenemos que escribir en la ventana del editor el siguiente programa

src/HolaMundo.cpp
```
1  #include <iostream>
2
3  using namespace std;
4
5  int main () {
6    cout << "Hola mundo" << endl;
7
8    return 0;
9  }
10
```

Si no queremos obtener un ***warning*** (***advertencia***) cuando compilemos el programa, debemos incluir una línea en blanco al final del fichero. Una vez escrito el programa, lo guardamos (**Ctrl + s**). Veamos un poco en qué consiste este programa. La **línea 1** incluye la ***biblioteca para la entrada y salida estándar***, que contiene las definiciones de ***cin*** y ***cout*** que nos van a permitir leer datos del teclado y escribir datos en la pantalla, respectivamente. En la **línea 3** indicamos que vamos a usar el ***espacio de nombres std*** (más adelante explicaremos por qué incluimos esta línea). En la **línea 5** tenemos la función ***main***, cuyo contenido se ejecuta cuando ejecutamos (valga la redundancia) nuestro programa (para qué sirven los argumentos de esta función también lo veremos más adelante). En la **línea 6** mostramos el mensaje **"Hola mundo"** por pantalla. Finalmente en la **línea 8** hacemos un ***return*** con el valor **0** para indicar que la ejecución ha transcurrido correctamente y no se ha producido ningún error (justo a la izquierda de ***main*** podemos ver cómo esta función devuelve un valor entero).

Una vez que ya sabemos qué es lo que hace este programa, veamos cómo se compila. Sin cerrar el editor que contiene el código, nos vamos a la consola que tenemos abierta. Veremos que está ***bloqueada*** (no podemos escribir nada en ella). El motivo es que anteriormente ejecutamos el ***kwrite en primer plano***, por lo que hasta

que no lo cerremos no podremos volver a introducir ninguna nueva orden en la consola. ¿Cómo podemos pasarlo a un ***segundo plano*** para que podamos seguir introduciendo órdenes en la consola sin tener que cerrar la ventana del ***kwrite***? El método es muy sencillo. Primero vamos a suspender el proceso asociado al ***kwrite***. Para ello, vamos a la consola y pulsamos **Ctrl+Z**.

```
^Z
[1]+  Stopped                 kwrite src/HolaMundo.cpp
profesor@DECSAI:~/mp$
```

Con esto, ya liberamos la consola para poder seguir escribiendo en ella. El problema ahora es que hemos bloqueado la ventana del ***kwrite*** (si intentamos escribir algo en ella veremos que no es posible) ya que hemos ***suspendido*** el proceso. Para que continúe ejecutándose pero en un segundo plano, tenemos que ver primero cuál es el ***identificador*** que tiene asociado. Esto lo podemos hacer tecleando la orden ***jobs*** en la consola.

```
profesor@DECSAI:~/mp$ jobs
[1]+  Stopped                 kwrite src/HolaMundo.cpp
profesor@DECSAI:~/mp$
```

Así podemos ver que la orden ***kwrite src/HolaMundo.cpp*** está suspendida y tiene asociada el identificador **1**. Para reanudar la ejecución pero en un ***segundo plano***, escribimos ***bg %1***, siendo **1** el identificador del proceso que queremos que pase a ejecutarse en ***segundo plano*** (***bg*** viene del inglés ***background***, que significa ***segundo plano***).

```
profesor@DECSAI:~/mp$ bg %1
[1]+ kwrite src/HolaMundo.cpp &
profesor@DECSAI:~/mp$
```

Así ya hemos conseguido lo que buscábamos. Por un lado podemos seguir introduciendo órdenes en la consola y por otro podemos seguir escribiendo en la ventana del editor. Si en lugar de escribir ***bg %1*** hubiésemos escrito ***fg %1*** (del inglés ***foreground***, o ***primer plano***) se hubiese reanudado la ejecución del editor pero en ***primer plano***, por lo que volveríamos a estar en la misma situación que al principio (no podríamos seguir escribiendo órdenes en la consola). ¿Cómo podemos evitar todo esto para que no tengamos que estar suspendiendo procesos y pasándolos a un ***segundo plano*** cada vez que queremos editar un archivo? La respuesta es muy sencilla, añadimos un ampersand (**&**) al final de la orden para que esta se ejecute directamente en un segundo plano. Así, abriremos los ficheros con el editor de la siguiente manera, ***kwrite nombreDelArchivo&*** (**&** puede ir pegado al nombre del archivo o separado, es indiferente).

```
profesor@DECSAI:~/mp$ kwrite src/HolaMundo.cpp&
[1] 7197
profesor@DECSAI:~/mp$
```

Ya que hemos retomado el control de la consola, vamos a ver cómo podemos ***compilar el programa*** para generar nuestro ***primer ejecutable***. Para ***compilar*** (en el amplio sentido de la palabra, es decir, todo lo que tiene que ver con el **preprocesador**, el **compilador** y el **enlazador**) vamos a usar la orden ***g++***, que es el compilador de GNU/Linux. Para crear en el directorio **bin** el ejecutable de nuestro programa, que se va a llamar **HolaMundo**, escribimos en la consola ***g++ -o bin/HolaMundo src/HolaMundo.cpp***.

```
profesor@DECSAI:~/mp$ g++ -o bin/HolaMundo src/HolaMundo.cpp
profesor@DECSAI:~/mp$
```

El formato va a ser siempre el mismo, ***g++ -o SALIDA ENTRADA***. También se puede compilar sin especificar el fichero de salida, es decir, el nombre del ejecutable, con lo que ***g++ ENTRADA*** generaría un fichero de nombre **a.out** que sería nuestro fichero ejecutable. Si todo ha salido bien (no hay ningún error en nuestro programa), tendremos nuestro ejecutable en la carpeta **bin**. ¿Cómo se lanza el ejecutable desde la línea de comandos? Muy fácil, escribimos ***./bin/HolaMundo***.

```
profesor@DECSAI:~/mp$ ./bin/HolaMundo
Hola mundo
profesor@DECSAI:~/mp$
```

El resultado es que se nos muestra por pantalla el mensaje **"Hola Mundo"**. Ya tenemos nuestro primer programa funcionando perfectamente.

Antes de pasar al siguiente programa, hagamos un pequeño inciso. En Linux, tanto los ficheros como los directorios tienen asociados una serie de ***permisos***. Estos permisos son de **lectura**, de **escritura**, y de **ejecución**. Además, estos tres tipos de permisos se pueden especificar para tres conjuntos de usuarios distintos, de forma que cada uno puede tener unos diferentes. Por un lado tenemos los permisos asociados al **propietario** del archivo (el usuario), por otro los permisos asociados a los usuarios que pertenecen al mismo **grupo** al que pertenece el usuario, y finalmente los permisos asociados al **resto** de usuarios. ¿Qué pasaría si una vez que hemos compilado nuestro programa no tuviésemos permiso de ejecución sobre el mismo? La respuesta es evidente, no podríamos ejecutar el programa.

```
profesor@DECSAI:~/mp$ ./bin/HolaMundo
bash: ./bin/HolaMundo: Permiso denegado
profesor@DECSAI:~/mp$
```

Ahora viene la parte que nos interesa, ¿cómo podemos ***cambiar los permisos*** de los archivos? Para ello usaremos el comando ***chmod***. El formato es el siguiente: ***chmod tipoDeUsuario±tipoDePermiso fichero***, donde **tipoDeUsuario** puede ser **u** (queremos modificar sólo los permisos del ***propietario*** del fichero), **g** (para los permisos del ***grupo***), **o** (para el ***resto*** de los usuarios, los que pertenecen a otros grupos), o **a** (para modificar ***los tres a la vez***, es decir, el del usuario, su grupo, y el resto de grupos), y **tipoDePermiso** puede ser **r** (***lectura***), **w** (***escritura***), o **x** (***ejecución***). Por lo tanto, para que podamos ejecutar nuestro programa escribimos en la línea de comandos ***chmod u+x bin/HolaMundo***.

```
profesor@DECSAI:~/mp$ chmod u+x bin/HolaMundo
profesor@DECSAI:~/mp$
```

Si lo que quisiésemos es ***quitarle el permiso*** en lugar de ponérselo, utilizaríamos un **-** (signo menos) en lugar de un **+** (signo más). Otra forma de modificar los permisos es cambiando ***tipoDeUsuario±tipoDePermiso*** por un número de tres cifras del cero al siete, cada una de las cuales equivale a tres bits que representan los permisos concedidos a ese tipo de usuario. Por ejemplo, el **777** equivale al número binario **111 111 111**, que indica que se conceden ***todos los permisos*** a todos los tipos de usuarios. Ejecutando ***chmod 744 bin/HolaMundo*** asignaríamos al propietario todos los permisos sobre el archivo, y a los usuarios del mismo grupo y al resto de usuarios les permitiríamos leer el archivo pero no escribir en él ni ejecutarlo.

u (usuario)			g (grupo)			o (resto)		
r (2^2)	w (2^1)	x (2^0)	r (2^2)	w (2^1)	x (2^0)	r (2^2)	w (2^1)	x (2^0)
1	1	1	1	0	0	1	0	0
7			4			4		

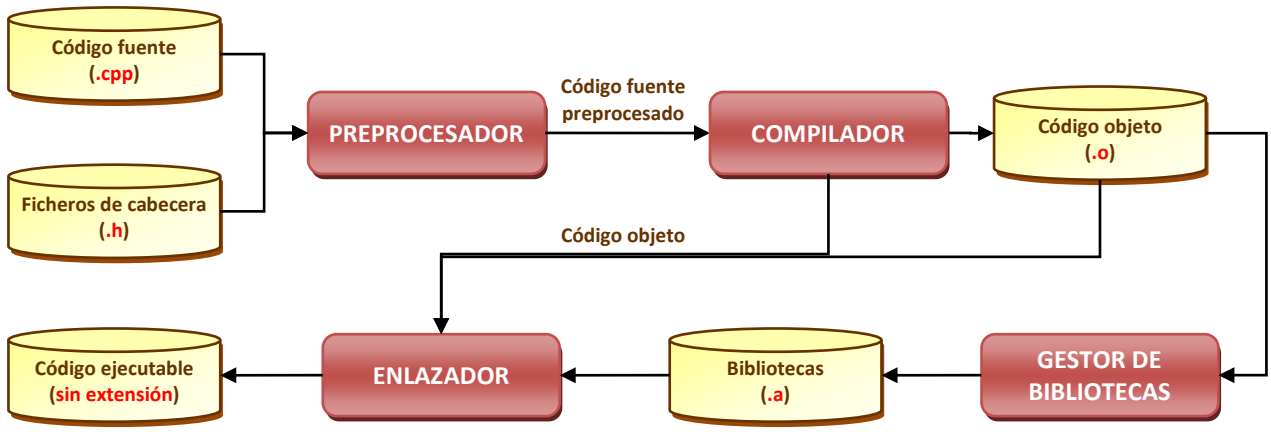
Además de cambiar los permisos, también podríamos ***cambiar el propietario del archivo*** usando el comando ***chown***, aunque no vamos a entrar en detalle (para más información podemos consultar la ayuda con ***man chown***). Con todo esto, ya no deberíamos tener problemas con la ejecución de nuestros programas.

¿ Qué ocurre cuando cuando hacemos ***g++ -o nombreFicheroEjecutable nombreFicheroCpp?***.

```
profesor@DECSAI:~/mp$ g++ -o bin/HolaMundo src/HolaMundo.cpp
profesor@DECSAI:~/mp$
```

El código fuente pasa primero por el **preprocesador**, que se encarga de eliminar los comentarios, e interpretar y procesar las ***directivas de preprocesamiento***, que van precedidas por el símbolo **#**. Si por ejemplo tenemos un ***#include <iostream>*** se cambia esa línea por el contenido del propio fichero ***iostream***, que contiene

declaraciones de tipos y funciones de entrada/salida de la biblioteca estándar de C++. Además, si hubiésemos definido algún ***identificador simbólico*** como por ejemplo ***#define MAX 100***, el **preprocesador** cambiaría cualquier aparición de ***MAX*** en el código por el literal ***100***, de tipo ***int***, a no ser que fuese una cadena de caracteres, es decir, **"MAX"**. El código fuente preprocesado, el cual no es almacenado en ningún fichero en disco, pasa ahora a manos del **compilador**, que se encarga de ***analizar su sintaxis y su semántica, y traducirlo***, generando un fichero que contiene el código objeto (**.o**). Finalmente, el **enlazador** ***resuelve las referencias a objetos externos***, es decir, objetos que se encuentran en otros módulos compilados (otros **.o** o **bibliotecas** (**.a**), las cuales son generadas por el **gestor de bibliotecas** a partir de una serie de ficheros **.o**) generando así un fichero ejecutable.



Como vemos en nuestro ejemplo, con una sola orden se puede llevar a cabo todo este proceso de **preprocesamiento**, **compilación** y **enlace**, pasando del código fuente **src/HolaMundo2.cpp** al ejecutable **bin/HolaMundo2**. En este caso, el fichero objeto no se conserva después del enlace, es solamente un fichero temporal. Sin embargo, también podemos hacer el proceso en dos fases. Primero preprocesamos y compilamos el código fuente para generar el código objeto (**.o**), y después enlazamos todos nuestros módulos compilados para crear el ejecutable. Para que ***g++*** no realice el **enlace**, es decir, que sólo haga el **preprocesamiento** y la **compilación**, tenemos que pasarle como argumento la opción **-c**. De esta forma, la salida será un fichero **.o** en lugar de un ejecutable. El formato sería ***g++ -c -o nombreFicheroO nombreFicheroCpp***.

```
profesor@DECSAI:~/mp$ g++ -c -o obj/HolaMundo2.o src/HolaMundo2.cpp
profesor@DECSAI:~/mp$
```

Y ahora ya ***enlazaríamos*** todos nuestros módulos compilados para crear el ejecutable (ya no hay que incluir la la opción **-c**).

```
profesor@DECSAI:~/mp$ g++ -o bin/HolaMundo2 obj/HolaMundo2.o
profesor@DECSAI:~/mp$
```

Podemos ejecutar el programa para comprobar que funciona correctamente.

```
profesor@DECSAI:~/mp$ ./bin/HolaMundo2
Hola Mundo
profesor@DECSAI:~/mp$
```

Ahora que ya hemos visto cómo se crea un programa muy sencillo y cómo se compila para generar el ejecutable correspondiente, vamos a incluir una serie de ejemplos en los que poco a poco iremos dividiendo el código fuente en varios ficheros, de forma que cada uno se compilará por separado obteniendo así diferentes ficheros objeto que posteriormente enlazaremos para crear los ejecutables.

Vamos a crear el primer programa llamado **demo1.cpp**, como siempre en el directorio **src**.

```
profesor@DECSAI:~/mp$ kwrite src/demo1.cpp&
[6] 7490
profesor@DECSAI:~/mp$
```

Este programa va a contener una función ***main***, que va a llamar a otras cuatro funciones, que se van a encargar de ***sumar***, ***restar***, ***multiplicar*** y ***dividir*** dos números enteros. El contenido del fichero debe ser el siguiente:

src/demo1.cpp

```
1 #include <iostream>
2
3 int suma (int a, int b) {
4   return a + b;
5 }
6
7 int resta (int a, int b) {
8   return a - b;
9 }
10
11 int multiplica (int a, int b) {
12   return a * b;
13 }
14
15 int divide (int a, int b) {
16   return a / b;
17 }
18
19 using namespace std;
20
21 int main (int argc, char *argv[]) {
22   int a, b;
23
24   cout << "Introduce el primer valor: ";
25   cin >> a;
26
27   cout << "Introduce el segundo valor: ";
28   cin >> b;
29
30   cout << "suma(" << a << ", " << b << ") = " << suma(a,b) << endl;
31   cout << "resta(" << a << ", " << b << ") = " << resta(a,b) << endl;
32   cout << "multiplica(" << a << ", " << b << ") = " << multiplica(a,b) << endl;
33   cout << "divide(" << a << ", " << b << ") = " << divide(a,b) << endl;
34
35   return 0;
36 }
37
```

Ahora lo compilamos.

```
profesor@DECSAI:~/mp$ g++ -o bin/demo1 src/demo1.cpp
profesor@DECSAI:~/mp$
```

Y lo ejecutamos.

```
profesor@DECSAI:~/mp$ ./bin/demo1
Introduce el primer valor: 10
Introduce el segundo valor: 2
suma(10, 2) = 12
resta(10, 2) = 8
multiplica(10, 2) = 20
divide(10, 2) = 5
profesor@DECSAI:~/mp$
```

En este caso tenemos ***un solo fichero con código fuente*** que contiene tanto la función ***main*** como todas las otras funciones que hemos creado. Esto no es lo más habitual ni lo más aconsejable, sino que lo que se suele hacer es **modularizar el software**, es decir, ***dividir los programas en varios ficheros***, lo cual aporta muchas

ventajas. Por un lado, cada uno de esos nuevos ficheros contendrá solamente ***funciones que estén relacionadas entre sí*** desde un punto de vista lógico. Por ejemplo, podemos tener un fichero con funciones para el tratamiento de cadenas de caracteres, otro para manipular números complejos, etc. Además, al encontrarse en ficheros diferentes al de la función ***main***, podemos ***reutilizar nuestras funciones*** en distintos programas, sin tener que copiar el mismo código en cada uno de ellos. Así, si detectamos un error en alguna función, solamente tenemos que cambiar el código de un solo fichero, y no cada uno de los fragmentos duplicados, en cuyo caso se nos podría olvidar alguno. Otra gran ventaja es el ***ocultamiento de información***. Si utilizamos un módulo objeto (**.o**) o una **biblioteca** (**.a**), no conocemos los detalles de implementación de las funciones definidas en estos, sino que sólo conocemos una interfaz que nos dice cómo podemos usar esas funciones. Esta interfaz se proporciona a través de los ficheros de cabecera (**.h**). Por estas y otras ventajas, la **modularización del software** es muy importante.

Tras esta pequeña aclaración, retomemos nuestro ejemplo. Como podemos ver, hemos definido las cuatro funciones ***suma***, ***resta***, ***multiplica*** y ***divide*** antes de la función ***main***. Este orden es importante ya que cuando el **compilador** llega al trozo de código de la función ***main*** en el que se realiza la llamada a esas cuatro funciones, este ya tiene que saber que esas funciones existen, por lo que si las ponemos después de la función ***main*** (al final del archivo) obtendremos el siguiente error en tiempo de compilación.

```
profesor@DECSAI:~/mp$ g++ -o bin/demo1 src/demo1.cpp
src/demo1.cpp: In function ‘int main(int, char**)’:
src/demo1.cpp:13: error: ‘suma’ no se declaró en este ámbito
src/demo1.cpp:14: error: ‘resta’ no se declaró en este ámbito
src/demo1.cpp:15: error: ‘multiplica’ no se declaró en este ámbito
src/demo1.cpp:16: error: ‘divide’ no se declaró en este ámbito
profesor@DECSAI:~/mp$
```

Podríamos colocar las cuatro funciones al final del archivo si las declaramos antes de la función ***main***. Para declararlas, incluimos el **prototipo** de cada una de ellas. Por ejemplo, el prototipo de la función ***suma*** es ***int suma (int a, int b);***, el cual especifica el nombre de la función, cuántos argumentos tiene y de qué tipo son (el nombre que tengan en realidad no haría falta ponerlo), y qué tipo de dato devuelve (si es que devuelve algún dato como en este caso). Así la función ***suma*** recibe dos números enteros y devuelve otro entero.

src/demo1.cpp

```
1  #include <iostream>
2
3  int suma (int a, int b);
4  int resta (int a, int b);
5  int multiplica (int a, int b);
6  int divide (int a, int b);
7
8  using namespace std;
9
10 int main (int argc, char *argv[]) {
11   int a, b;
12
13   cout << "Introduce el primer valor: ";
14   cin >> a;
15
16   cout << "Introduce el segundo valor: ";
17   cin >> b;
18
19   cout << "suma(" << a << ", " << b << ") = " << suma(a,b) << endl;
20   cout << "resta(" << a << ", " << b << ") = " << resta(a,b) << endl;
21   cout << "multiplica(" << a << ", " << b << ") = " << multiplica(a,b) << endl;
22   cout << "divide(" << a << ", " << b << ") = " << divide(a,b) << endl;
23
24   return 0;
25 }
26
27 int suma (int a, int b) {
28   return a + b;
29 }
30
31 int resta (int a, int b) {
32   return a - b;
```

```
33 }
34
35 int multiplica (int a, int b) {
36   return a * b;
37 }
38
39 int divide (int a, int b) {
40   return a / b;
41 }
42
```

Este trozo de código ya sí compila perfectamente.

```
profesor@DECSAI:~/mp$ g++ -o bin/demo1 src/demo1.cpp
profesor@DECSAI:~/mp$
```

Ahora vamos a empezar con la **modularización del programa**. Primero realizamos una copia del código fuente anterior ya que vamos a realizar unas pequeñas modificaciones. Creamos por tanto el fichero **demo2.cpp** utilizando el comando ***cp***, cuyo formato es ***cp ORIGEN DESTINO***.

```
profesor@DECSAI:~/mp$ cp src/demo1.cpp src/demo2.cpp
profesor@DECSAI:~/mp$
```

Ahora editamos el fichero que acabamos de crear.

```
profesor@DECSAI:~/mp$ kwrite src/demo2.cpp&
[8] 7510
profesor@DECSAI:~/mp$
```

Vamos a crear también un fichero en el que pondremos el código fuente de nuestras cuatro funciones. Como vamos a tener muchas versiones de cada fichero, para que quede un poco más claro incluiremos, al final de su nombre, el ***número de versión del programa principal***. Por ejemplo, como este programa se llama **demo2.cpp**, el fichero que contendrá el código de las funciones se llamará **oper2.cpp** (aunque no exista ningún **oper.cpp**).

```
profesor@DECSAI:~/mp$ kwrite src/oper2.cpp&
[9] 7511
profesor@DECSAI:~/mp$
```

También creamos un fichero de cabecera llamado **oper2.h** para almacenar los prototipos de nuestras funciones. Al ser un fichero de cabecera lo crearemos en el directorio **include**.

```
profesor@DECSAI:~/mp$ kwrite include/oper2.h&
[10] 7512
profesor@DECSAI:~/mp$
```

Ahora vamos a rellenar los ficheros que hemos creado. En el fichero de cabecera, **include/oper2.h**, dejaremos solamente los prototipos de las cuatro funciones.

include/oper2.h

```
1 #ifndef OPER2
2 #define OPER2
3 int suma (int a, int b);
4 int resta (int a, int b);
5 int multiplica (int a, int b);
6 int divide (int a, int b);
7 #endif
8
```

Observemos que además de los prototipos de las cuatro funciones hemos incluido otras tres líneas. En la **línea 1**, la directiva ***#ifndef OPER2*** comprueba la existencia del identificador **OPER2** (el nombre del fichero de cabecera en mayúsculas, pero sin el **.h**). Este existirá si previamente se ha utilizado en una macrodefinición

(con la cláusula ***#define***). ¿Por qué es aconsejable incluir esta línea? Para prevenir la inclusión repetida de un fichero de cabecera en un mismo fichero fuente. Aunque parece lógico que nadie va a realizar un mismo ***#include*** dos veces en un mismo fichero:

```
#include "cabecera1.h"
#include "cabecera1.h"
```

sí puede ocurrir que la inclusión sea transitiva, es decir, que incluyamos dos ficheros de cabecera uno de los cuales contiene a su vez un ***#include*** al otro:

```
#include "cabecera1.h"
#include "cabecera2.h"
```

De esta forma, si **cabecera2.h** contiene un ***#include "cabecera1.h"***, el contenido de **cabecera1.h** se estaría copiando dos veces. Esto provocaría ***errores por definición múltiple***. Por esa razón se protege el contenido del fichero de cabecera con las directivas **#ifndef … #define … #endif**, de manera que en el primer ***#include*** la constante simbólica **OPER2** no está definida, por lo que se define y se procesa (copia) el resto del fichero de cabecera. En el segundo ***#include***, la constante simbólica **OPER2** ya está definida, por lo que el preprocesador salta a la línea siguiente al predicado ***#endif*** (en este caso no se incluye nada ya que no hay nada después de este predicado).
Una vez que tenemos nuestro fichero de cabecera, vamos a rellenar el fichero con el código fuente de las cuatro funciones:

src/oper2.cpp

```
1  #include "oper2.h"
2
3  int suma (int a, int b) {
4    return a + b;
5  }
6
7  int resta (int a, int b) {
8    return a - b;
9  }
10
11 int multiplica (int a, int b) {
12   return a * b;
13 }
14
15 int divide (int a, int b) {
16   return a / b;
17 }
18
```

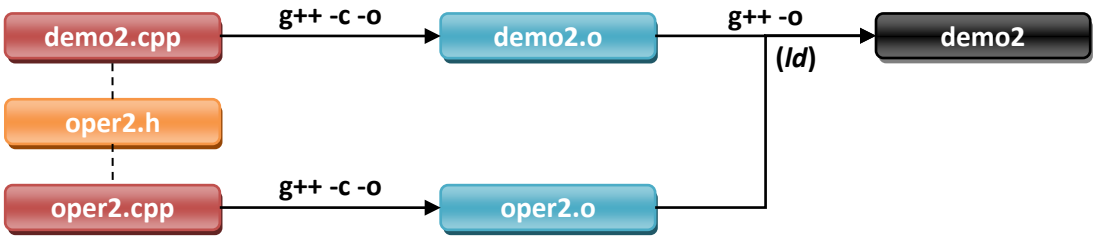
Además de las definiciones de las funciones, hemos incluido el fichero de cabecera con los prototipos de las mismas. Observemos que en este caso el nombre del fichero de cabecera va entre comillas dobles (**"oper2.h"**), cuando para ***iostream***, por ejemplo, se utilizaban los caracteres mayor que y menor que (**<*iostream*>**). La diferencia es que con ***#include <iostream>*** el fichero se busca en los directorios de ficheros de cabecera del sistema o en los directorios especificados con la opción **-I** (i mayúscula, de **Include**) del **compilador** (más adelante usaremos esta opción), mientras que con ***#include "oper2.h"*** el fichero se busca en el directorio donde se está realizando la compilación o en el especificado en el nombre del fichero.

Finalmente, el fichero **src/demo2.cpp** contendrá solamente la definición de la función ***main*** e incluirá el fichero de cabecera **oper2.h** que contiene los prototipos de las funciones ***suma***, ***resta***, ***multiplica*** y ***divide***, de las cuales solamente las dos primeras se utilizan en la función ***main***.

src/demo2.cpp

```
1 #include <iostream>
2 #include "oper2.h"
3
4 using namespace std;
5
6 int main (int argc, char *argv[]) {
7   int a, b;
8
9   cout << "Introduce el primer valor: ";
10  cin >> a;
11
12  cout << "Introduce el segundo valor: ";
13  cin >> b;
14
15  cout << "suma(" << a << ", " << b << ") = " << suma(a,b) << endl;
16  cout << "resta(" << a << ", " << b << ") = " << resta(a,b) << endl;
17
18  return 0;
19 }
20
```

¿Cómo se compilan ahora todos estos ficheros para generar un único ejecutable? El proceso es muy sencillo. Veámoslo gráficamente.



Primero vamos a generar el código objeto **demo2.o** a partir del código fuente **demo2.cpp**, el cual incluye el fichero de cabecera **oper2.h**. Para ello, compilamos con ***g++*** especificando la opción **-c**, de forma que todavía no se pasa el **.o** al enlazador para generar el ejecutable. De hecho no podemos crear el ejecutable sólo a partir de **demo2.cpp** ya que nos haría falta el código de **oper2.cpp**.

```
profesor@DECSAI:~/mp$ g++ -c -o obj/demo2.o src/demo2.cpp
src/demo2.cpp:2:19: error: oper2.h: No existe el fichero ó directorio
src/demo2.cpp: In function ‘int main(int, char**)’:
src/demo2.cpp:15: error: ‘suma’ no se declaró en este ámbito
src/demo2.cpp:16: error: ‘resta’ no se declaró en este ámbito
profesor@DECSAI:~/mp$
```

Como vemos, el compilador no es capaz de encontrar el fichero **oper2.h**, por lo que no reconoce las funciones ***suma*** y ***resta***. Tenemos que especificar, con la opción **-I** (i mayúscula), el directorio en el que se encuentran los ficheros de cabecera.

```
profesor@DECSAI:~/mp$ g++ -c -o obj/demo2.o src/demo2.cpp -I./include
profesor@DECSAI:~/mp$
```

A continuación tenemos que generar el código objeto **oper2.o** asociado a **oper2.cpp**, el cual también incluye el fichero de cabecera **oper2.h**.

```
profesor@DECSAI:~/mp$ g++ -c -o obj/oper2.o src/oper2.cpp -I./include
profesor@DECSAI:~/mp$
```

Finalmente, enlazamos los dos ficheros con código objeto para generar el ejecutable **demo2**. Recordemos que para que se pueda crear el ejecutable uno de ellos tiene que contener una función ***main*** (en nuestro caso está contenida en **demo2.o**).

```
profesor@DECSAI:~/mp$ g++ -o bin/demo2 obj/demo2.o obj/oper2.o
profesor@DECSAI:~/mp$
```

Como vemos, si representamos gráficamente nuestros ficheros y las dependencias entre ellos después es muy sencillo el proceso de compilación, por lo que esta es la metodología que siempre seguiremos. Sin embargo, ya vemos como poco a poco la cosa se va complicando (en este ejemplo hemos llamado tres veces a ***g++*** para generar nuestro ejecutable).

La pregunta que surge ahora es si todo este proceso de compilación se puede hacer ***de forma automática***. La respuesta es que sí. Para ello vamos a escribir en ficheros **makefile** las reglas que crean los diferentes módulos objeto y los ejecutables. Así, ejecutando el programa ***make*** se compilarán de forma automática todos los ficheros que hagan falta (si no se han modificado no se tienen que volver a compilar de nuevo) para crear nuestro ejecutable. Lo primero que vamos a ver es qué formato tienen este tipo de ficheros. Vamos a partir del ejemplo más simple, **demo1.cpp** (que contenía todo el código fuente en un solo archivo), para ver cómo se construye el ejecutable **demo1** empleando un fichero **makefile** que se va a llamar **makefile**.

```
profesor@DECSAI:~/mp$ kwrite makefile&
[11] 7783
profesor@DECSAI:~/mp$
```

Como vemos, este fichero lo creamos en la carpeta **mp**, sin incluirlo en ninguna subcarpeta. Copiamos ahora el siguiente texto en el fichero que acabamos de crear.

makefile

```
1 all : bin/demo1
2
3 bin/demo1 : obj/demo1.o
4 »         g++ -o bin/demo1 obj/demo1.o
5
6 obj/demo1.o : src/demo1.cpp
7 »         g++ -c -o obj/demo1.o src/demo1.cpp
8
```

Veamos cuál es su estructura. Aunque no es estrictamente necesario, en la **línea 1** siempre vamos a incluir un ***destino simbólico***, que llamaremos **all**. Es simbólico porque no lleva asociada ninguna orden como en los dos casos siguientes, que llevan asociada una orden ***g++***. Después del nombre del destino simbólico (**all**) ponemos el carácter dos puntos (**:**) seguido por una ***lista de destinos***, donde ponemos todos los ejecutables que queramos crear (en este caso vamos a crear sólo uno, **bin/demo1**). De esta forma, cuando invoquemos al programa ***make*** se construirán todos los destinos de esa lista, sin tener que invocarlo tantas veces como destinos queramos construir. Ahora tenemos que decirle a **make** cómo puede crear ese destino **bin/demo1**. Por ello, en la **línea 3** se escribe el nombre del destino, **bin/demo1**, seguido como siempre del carácter dos puntos (**:**) y a continuación la ***lista de dependencias*** (**obj/demo1.o** en este caso), que no es más que los ficheros (separados por espacios en blanco) de los que depende el destino. Así, estamos diciendo que **bin/demo1** depende de **obj/demo1.o**, y que una vez que tengamos el archivo **obj/demo1.o**, el ejecutable se crea con la orden de la **línea 4**. Una cosa muy importante que no se nos puede olvidar es que las órdenes empiezan con un **tabulador** (delante de los dos ***g++***). Si no hacemos esto, ***make*** mostrará un error y no continuará procesando el fichero **makefile**. Las **líneas 3 y 4** constituyen lo que se conoce como una ***regla explícita***. Es explícita porque somos nosotros los que estamos escribiendo a mano cuáles son exactamente las órdenes que crean todos y cada uno de los ejecutables y ficheros objeto. Sin embargo, al llegar a la ***lista de dependencias*** de la **línea 3**, ***make*** detecta que **obj/demo1.o** todavía no está creado, por lo que busca una regla para construirlo. Se está produciendo una reacción en cadena que se llama ***dependencia encadenada***. En la **línea 6** especificamos que el módulo objeto **obj/demo1.o** depende del archivo **src/demo1.cpp**, y por lo tanto, cada vez que se modifique **src/demo1.cpp** hay que volver a construir el destino **obj/demo1.o** utilizando la orden que aparece en la **línea 7**. Como vemos, el fichero **makefile** se escribe en el orden inverso a como teclearíamos las órdenes en la consola (primero especificamos cómo se crea el ejecutable, y luego cada uno de los códigos objetos de los que depende).

Ahora que ya tenemos creado nuestro **makefile**, para lanzarlo escribimos simplemente ***make*** en la línea de comandos.

```
profesor@DECSAI:~/mp$ make
g++ -c -o obj/demo1.o src/demo1.cpp
g++ -o bin/demo1 obj/demo1.o
profesor@DECSAI:~/mp$
```

Así podemos ver cómo se genera primero el fichero **obj/demo1.o** a partir de **src/demo1.cpp**, y a continuación se utiliza ese código objeto para generar el ejecutable **bin/demo1**. Si volvemos a ejecutar la orden ***make*** veremos cómo este detecta que ***no hace falta recompilar nada***, ya que los ficheros fuente no han sido modificados (***make*** compara las fechas de modificación de los ficheros para saber si estos han cambiado).

```
profesor@DECSAI:~/mp$ make
make: No se hace nada para `all'.
profesor@DECSAI:~/mp$
```

Hay que comentar que el nombre del fichero **makefile** no tiene por qué se **makefile**, sino que puede ser cualquier otro, en cuyo caso habría que ejecutar el programa ***make*** especificando con la opción **-f** el nombre del fichero **makefile** (si no se especifica esa opción, ***make*** sólo busca algún fichero de nombre **makefile** o **Makefile** en el directorio actual). El resultado que se produciría sería el mismo.

```
profesor@DECSAI:~/mp$ make -f makefile
make: No se hace nada para `all'.
profesor@DECSAI:~/mp$
```

Otra opción interesante de ***make***, **-n**, **--just-print**, **--dry-run**, o **--recon**, hace que ***no se ejecuten las instrucciones***, sino que solamente se muestren por pantalla. De esta manera, podemos ver si el fichero **makefile** que hemos escrito es correcto o no.

También podemos especificar como argumento el nombre del destino que queremos construir. Si no se especifica ninguno, ***make*** intenta construir solamente el primer destino.

Vamos a crear ahora un **makefile** para **demo2**. Lo llamaremos en este caso **make2**.

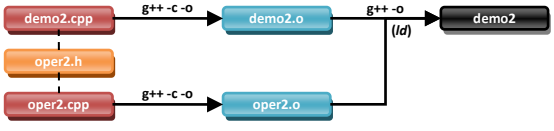
```
profesor@DECSAI:~/mp$ kwrite make2&
[12] 7811
profesor@DECSAI:~/mp$
```

El contenido de este fichero será el siguiente:

make2

```
1  all : bin/demo2
2
3  bin/demo2 : obj/demo2.o obj/oper2.o
4  »        g++ -o bin/demo2 obj/demo2.o obj/oper2.o
5
6  obj/demo2.o : src/demo2.cpp include/oper2.h
7  »        g++ -c -I./include -o obj/demo2.o src/demo2.cpp
8
9  obj/oper2.o : src/oper2.cpp include/oper2.h
10 »        g++ -c -I./include -o obj/oper2.o src/oper2.cpp
11
```

El **makefile** se puede crear muy fácilmente si nos fijamos en el gráfico de la derecha, en el que aparecen las dependencias entre los distintos módulos. El ejecutable que queremos crear es **bin/demo2**, por lo que se incluye en la



lista de destinos de la **línea 1**. Ahora tenemos que especificar cómo se construye ese *destino* (**línea 3**). Como vemos en el gráfico, este depende de **obj/demo2.o** y de **obj/oper2.o**, por lo que incluimos esos dos ficheros objeto en la *lista de dependencias* de esa regla. En la **línea 4** incluimos la *orden* que se encarga de crear el ejecutable a partir de los dos objetos. A continuación, en la **línea 6** indicamos cómo se construye **obj/demo2.o**, que depende de **src/demo2.cpp** e indirectamente de **include/oper2.h**, ya que **src/demo2.cpp** incluye dicho fichero de cabecera. Por lo tanto, si se modifica el fichero **include/oper2.h** hay que volver a generar **obj/demo2.o**. La orden *g++* asociada a esta regla, que aparece en la **línea 7**, incluye la opción **-I** (i mayúscula) para indicar dónde se tienen que buscar los ficheros de cabecera que hemos incluido en nuestro código. El formato de la opción era **-Idirectorio** (en nuestro caso, el directorio es **include**, que cuelga del directorio en el que se encuentra el fichero **make2**, de ahí el **./include**). Por último, en la **línea 8** indicamos cómo se construye **obj/oper2.o** a partir de **src/oper2.cpp** y **include/oper2.h**.

Otra cosa interesante que debemos conocer es que podemos incluir *comentarios* (aclaraciones) en el fichero **makefile** que indiquen lo que vamos haciendo en cada momento. Una línea de comentario empieza por el carácter almohadilla (**#**).

make2

```
1  # Fichero: make2
2  # Ejemplo de fichero makefile con un destino simbólico llamado "all"
3
4  all : bin/demo2
5
6  bin/demo2 : obj/demo2.o obj/oper2.o
7  »       echo Creando el ejecutable bin/demo2...
8  »       g++ -o bin/demo2 obj/demo2.o obj/oper2.o
9
10 obj/demo2.o : src/demo2.cpp include/oper2.h
11 »       @echo Creando el modulo objeto obj/demo2.o...
12 »       @g++ -c -I./include -o obj/demo2.o src/demo2.cpp
13
14 obj/oper2.o : src/oper2.cpp include/oper2.h
15 »       @echo Creando el modulo objeto obj/oper2.o...
16 »       @g++ -c -I./include -o obj/oper2.o src/oper2.cpp
17
```

También podemos incluir *más de una orden en cada regla*, cada una en una **línea distinta** y empezando con un **tabulador**. Se puede incluir cualquier orden que sea válida en el sistema operativo. Las órdenes pueden ir precedidas por **prefijos**. Los dos más importantes son el arroba (**@**), que *desactiva el eco* durante la ejecución de la orden, y el signo menos (**-**), que *ignora los errores* que puede producir la orden a la que precede. Por ejemplo, en las **líneas 7, 11 y 15** se han incluido tres órdenes *echo* que muestran mensajes por pantalla. En la primera (**línea 7**) no se ha incluido el prefijo **@**, mientras que en las otras dos sí. Por lo tanto, estas dos últimas no mostrarán esta orden por pantalla, sino solamente su resultado. También podemos poner el carácter **@** delante de las órdenes *g++*, como ocurre en las **líneas 12 y 16**. Si ejecutamos el **makefile** podemos ver qué es lo que ocurre exactamente.

```
profesor@DECSAI:~/mp$ make -f make2
Creando el modulo objeto obj/demo2.o...
Creando el modulo objeto obj/oper2.o...
echo Creando el ejecutable bin/demo2...
Creando el ejecutable bin/demo2...
g++ -o bin/demo2 obj/demo2.o obj/oper2.o
profesor@DECSAI:~/mp$
```

Como podemos ver, la primera orden *echo*, al no llevar el prefijo **@**, muestra por pantalla no sólo el mensaje asociado sino también la propia orden.

Si mostramos el contenido del directorio **obj**, veremos que este contiene todos los ficheros **.o** que hemos ido creando.

```
profesor@DECSAI:~/mp$ ls -la
total 48
drwxr-xr-x  8 profesor profesor 4096 2009-03-23 14:34 .
drwxr-xr-x 42 profesor profesor 4096 2009-03-23 14:02 ..
drwxr-xr-x  2 profesor profesor 4096 2009-03-23 14:35 bin
drwxr-xr-x  2 profesor profesor 4096 2009-03-23 13:58 doc
drwxr-xr-x  2 profesor profesor 4096 2009-03-23 14:31 include
drwxr-xr-x  2 profesor profesor 4096 2009-03-23 13:58 lib
-rw-r--r--  1 profesor profesor  508 2009-03-23 14:34 make2
-rw-r--r--  1 profesor profesor  508 2009-03-23 14:34 make2~
-rw-r--r--  1 profesor profesor  137 2009-03-23 14:31 makefile
-rw-r--r--  1 profesor profesor  137 2009-03-23 14:31 makefile~
drwxr-xr-x  2 profesor profesor 4096 2009-03-23 14:35 obj
drwxr-xr-x  2 profesor profesor 4096 2009-03-23 14:32 src
profesor@DECSAI:~/mp$
```

Sin embargo, podemos hacer que estos se ***eliminen de forma automática***. Para ello, vamos a incluir una nueva regla en el fichero **makefile**. El nombre del destino de esta nueva regla se llamará **clean**, mientras que la lista de dependencias estará vacía, ya que la construcción de ese destino no depende de nada.

make2

```
1  # Fichero: make2
2  # Ejemplo de fichero makefile con un destino simbólico llamado "all"
3
4  all : bin/demo2
5
6  bin/demo2 : obj/demo2.o obj/oper2.o
7  »         @echo Creando el ejecutable bin/demo2...
8  »         g++ -o bin/demo2 obj/demo2.o obj/oper2.o
9
10 obj/demo2.o : src/demo2.cpp include/oper2.h
11 »         @echo Creando el modulo objeto obj/demo2.o...
12 »         g++ -c -I./include -o obj/demo2.o src/demo2.cpp
13
14 obj/oper2.o : src/oper2.cpp include/oper2.h
15 »         @echo Creando el modulo objeto obj/oper2.o...
16 »         g++ -c -I./include -o obj/oper2.o src/oper2.cpp
17
18 clean :
19 »         @echo Borrando los ficheros .o del directorio obj...
20 »         rm obj/*.o
21
```

Para que ***make*** construya el destino **clean**, habrá que incluirlo en la llamada, ya que en caso contrario sólo intentaría construir el primer destino del fichero **makefile**, que en este caso es **all**.

```
profesor@DECSAI:~/mp$ make -f make2 clean
Borrando los ficheros .o del directorio obj...
rm obj/*.o
profesor@DECSAI:~/mp$
```

Para evitar tener que poner **clean** en cada llamada a ***make***, podemos incluir dicho destino en la lista de destinos asociados al destino simbólico **all** (**línea 1** del fichero **makefile**). De esta forma, siempre se intentará construir dicho destino.

make2

```
1  # Fichero: make2
2  # Ejemplo de fichero makefile con un destino simbólico llamado "all"
3
4  all : bin/demo2 clean
5
6  bin/demo2 : obj/demo2.o obj/oper2.o
7  »         @echo Creando el ejecutable bin/demo2...
8  »         g++ -o bin/demo2 obj/demo2.o obj/oper2.o
9
10 obj/demo2.o : src/demo2.cpp include/oper2.h
11 »         @echo Creando el modulo objeto obj/demo2.o...
12 »         g++ -c -I./include -o obj/demo2.o src/demo2.cpp
13
```

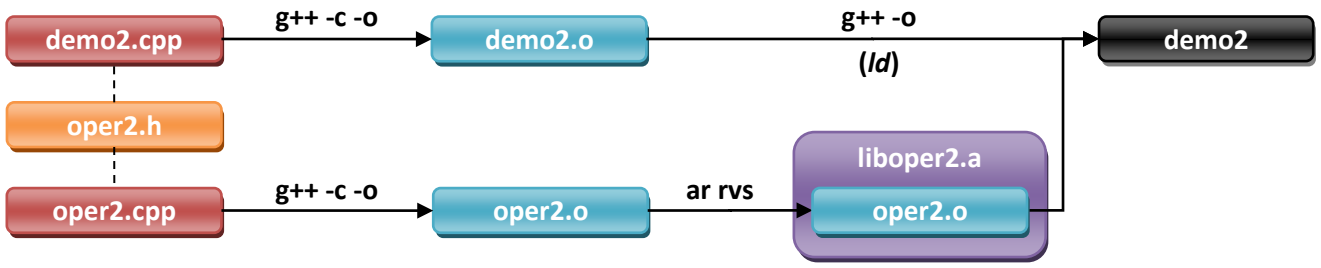
```
14 obj/oper2.o : src/oper2.cpp include/oper2.h
15 »       @echo Creando el modulo objeto obj/oper2.o...
16 »       g++ -c -I./include -o obj/oper2.o src/oper2.cpp
17
18 clean :
19 »       @echo Borrando los ficheros .o del directorio obj...
20 »       rm obj/*.o
21
```

Ahora ya ejecutaríamos el programa ***make*** especificando solamente el nombre del fichero **makefile**. El resultado que se produciría sería el mismo, solo que en este caso tenemos que volver a crear los módulos objeto porque los acabamos de borrar.

```
profesor@DECSAI:~/mp$ make -f make2
Creando el modulo objeto obj/demo2.o...
Creando el modulo objeto obj/oper2.o...
echo Creando el ejecutable bin/demo2...
Creando el ejecutable bin/demo2...
g++ -o bin/demo2 obj/demo2.o obj/oper2.o
Borrando los ficheros .o del directorio obj...
rm obj/*.o
profesor@DECSAI:~/mp$
```

Hay que aclarar que si el directorio actual contiene un fichero de nombre **clean**, la orden **make** no funcionará (no se borrarán los ficheros **.o**) ya que el destino **clean** no tiene ninguna dependencia y existe el fichero **clean**. Así, **make** supone que no tiene que volver a generar el fichero **clean**, ya que está actualizado, por lo que no ejecutaría la orden que borra los ficheros **.o**. Para solucionar esto, habría que declarar este tipo de destinos como falsos (**phony**) escribiendo **.PHONY : clean**. También sería conveniente hacer lo mismo para el destino simbólico **all**.

Lo siguiente que vamos a ver es cómo se crean las **bibliotecas** (del inglés ***libraries***) que son ficheros que ***agrupan un conjunto de módulos objeto*** (ficheros con extensión **.o**). Estas se pueden enlazar con algún código objeto que contenga una función ***main*** para crear un ejecutable. Si consideramos el ejecutable **bin/demo2**, este se crearía tal y como aparece en el siguiente gráfico.



Como vemos, la extensión por defecto de los ficheros de **biblioteca** es **.a**, y suelen comenzar con la palabra **lib**. En nuestro ejemplo la **biblioteca** se llamará **liboper2.a** y la guardaremos en el directorio **lib**. También hay otras **bibliotecas** ya creadas como **libm.a**, la cual contiene muchas funciones matemáticas como por ejemplo la del cálculo del seno. Las **bibliotecas** se crean con el programa ***ar***, especificando como argumento los caracteres **rvs**, donde **r** indica el ***tipo de operación*** a realizar (añade o reemplaza el módulo objeto de la biblioteca por la nueva versión), **v** sirve para ***mostrar información sobre la operación*** realizada, y **s** ***crea o actualiza el índice de los módulos*** que componen la **biblioteca**, de forma que el **enlazador** sepa cómo enlazarla. Para poder trabajar a mano con los **.o** tenemos que quitar momentáneamente el destino **clean** del **makefile** anterior.

```
profesor@DECSAI:~/mp$ ar rvs lib/liboper2.a obj/oper2.o
ar: creating lib/liboper2.a
a - obj/oper2.o
profesor@DECSAI:~/mp$
```

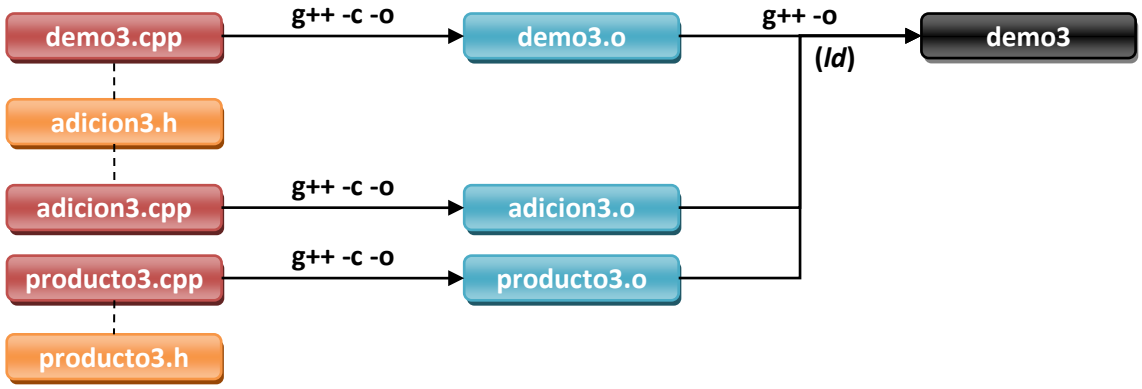
De esta forma creamos la **biblioteca** **lib/liboper2.a** que contiene solamente un módulo objeto, **obj/oper2.o**. Para construir el ejecutable a partir de ella usamos la opción **-l** ('*ele*' minúscula, de ***library***) de ***g++***, seguida del nombre de la **biblioteca** quitando el **lib** del principio y el **.a** del final. Así en nuestro caso será **-loper2**. Es importante destacar que, al igual que teníamos que usar la opción **-I** (i mayúscula) para especificar en qué directorio estaban los ficheros de cabecera (**.h**), ahora tenemos que usar la opción **-L** para especificar dónde hay que buscar la **biblioteca**, que para nuestro ejemplo será el directorio **lib**.

```
profesor@DECSAI:~/mp$ g++ -o bin/demo2 obj/demo2.o -L./lib -loper2
profesor@DECSAI:~/mp$
```

Así, ***g++*** (haciendo uso internamente de ***ld***, que es el programa enlazador proporcionado por GNU) enlaza el código asociado a **oper2.o** con el de **demo2.o** (que contiene la función ***main***) para generar el ejecutable **demo2**.

El programa ***ar*** tiene otras opciones interesantes: **d** ***elimina un módulo objeto*** (**.o**) de la **biblioteca**, **x** ***extrae un módulo objeto*** de la **biblioteca** manteniendo la biblioteca inalterada, y **t** ***lista los módulos objeto*** que componen la **biblioteca**.

Para practicar con todo lo que hemos aprendido vamos a crear un nuevo programa, **demo3.cpp**. Ahora la idea, tal y como se muestra en el siguiente gráfico, es dividir el fichero **oper2.cpp** en dos, de forma que tendremos un fichero **adicion3.cpp** con el código fuente de las funciones ***suma*** y ***resta***, y otro fichero llamado **producto3.cpp** con el código de las funciones ***multiplica*** y ***divide***. Cada uno tendrá su propio fichero de cabecera.



Primero creamos el fichero **src/demo3.cpp** que va a contener la función ***main***.

```
profesor@DECSAI:~/mp$ kwrite src/demo3.cpp&
[13] 7971
profesor@DECSAI:~/mp$
```

El contenido de este fichero es el mismo que el de **demo2.cpp**, solo que en este caso el fichero de cabecera que contiene las declaraciones de las funciones ***suma*** y ***resta*** es ahora **adicion3.h** en lugar de **oper2.h**.

src/demo3.cpp

```
1  #include <iostream>
2  #include "adicion3.h"
3
4  using namespace std;
5
6  int main (int argc, char *argv[]) {
7    int a, b;
8
9    cout << "Introduce el primer valor: ";
10   cin >> a;
11
12   cout << "Introduce el segundo valor: ";
13   cin >> b;
```

```
14
15   cout << "suma(" << a << ", " << b << ") = " << suma(a,b) << endl;
16   cout << "resta(" << a << ", " << b << ") = " << resta(a,b) << endl;
17
18   return 0;
19 }
20
```

Ahora creamos el fichero **include/adicion3.h** que va a contener las cabeceras de las funciones ***suma*** y ***resta***.

```
profesor@DECSAI:~/mp$ kwrite include/adicion3.h&
[14] 7973
profesor@DECSAI:~/mp$
```

Incluimos el siguiente código en el fichero que acabamos de crear.

include/adicion3.h

```
1 int suma (int a, int b);
2 int resta (int a, int b);
3
```

La definición de estas funciones se incluye en el fichero **src/adicion3.cpp**.

```
profesor@DECSAI:~/mp$ kwrite src/adicion3.cpp&
[15] 7975
profesor@DECSAI:~/mp$
```

src/adicion3.cpp

```
1 #include "adicion3.h"
2
3 int suma (int a, int b) {
4   return a + b;
5 }
6
7 int resta (int a, int b) {
8   return a - b;
9 }
10
```

Repetimos el proceso para los ficheros **include/producto3.h** y **src/producto3.cpp**, que contienen las declaraciones y las definiciones de las funciones ***multiplica*** y ***divide***.

```
profesor@DECSAI:~/mp$ kwrite include/producto3.h&
[16] 7976
profesor@DECSAI:~/mp$
```

include/producto3.h

```
1 int multiplica (int a, int b);
2 int divide (int a, int b);
3
```

```
profesor@DECSAI:~/mp$ kwrite src/producto3.cpp&
[17] 7977
profesor@DECSAI:~/mp$
```

src/producto3.cpp

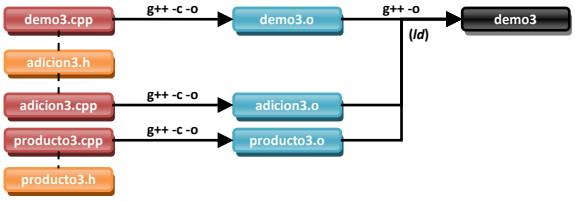
```
1 #include "producto3.h"
2
3 int multiplica (int a, int b) {
4   return a * b;
5 }
6
7 int divide (int a, int b) {
8   return a / b;
9 }
10
```

Ahora sólo nos queda crear el fichero **makefile** para que se compile todo de forma automática, y no tengamos que estar tecleando todas las órdenes una por una. A este fichero lo vamos a llamar **make3**.

```
profesor@DECSAI:~/mp$ kwrite make3&
[18] 7978
profesor@DECSAI:~/mp$
```

make3

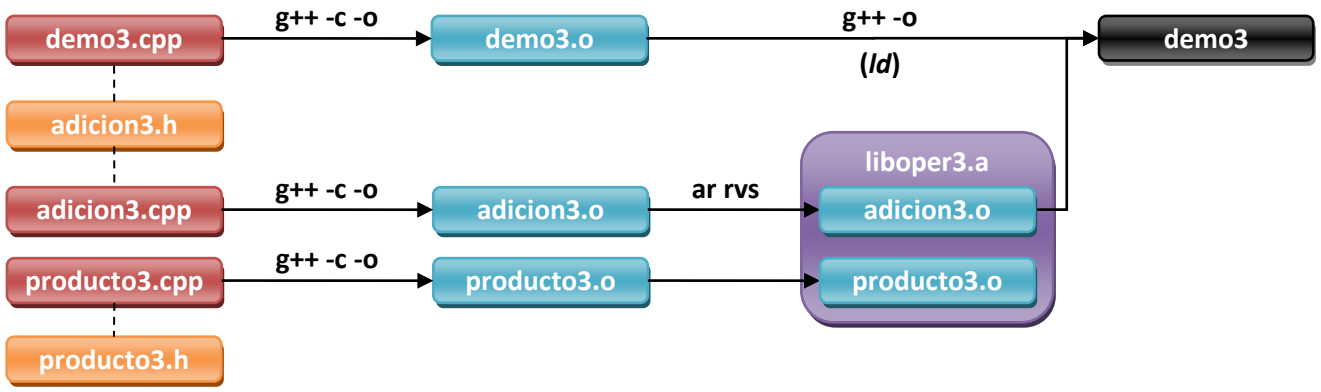
```
1  all : bin/demo3 clean
2
3  bin/demo3 : obj/demo3.o obj/adicion3.o obj/producto3.o
4  »         @echo Creando el ejecutable bin/demo3...
5  »         g++ -o bin/demo3 obj/demo3.o obj/adicion3.o obj/producto3.o
6
7  obj/demo3.o : src/demo3.cpp include/adicion3.h
8  »         @echo Creando el modulo objeto obj/demo3.o...
9  »         g++ -c -I./include -o obj/demo3.o src/demo3.cpp
10
11 obj/adicion3.o : src/adicion3.cpp include/adicion3.h
12 »         @echo Creando el modulo objeto obj/adicion3.o...
13 »         g++ -c -I./include -o obj/adicion3.o src/adicion3.cpp
14
15 obj/producto3.o : src/producto3.cpp include/producto3.h
16 »         @echo Creando el modulo objeto obj/producto3.o...
17 »         g++ -c -I./include -o obj/producto3.o src/producto3.cpp
18
19 clean :
20 »         @echo Borrando los ficheros .o del directorio obj...
21 »         rm obj/*.o
22
```



Como siempre, vamos a fijarnos en el gráfico que muestra nuestros ficheros y las dependencias entre ellos para que nos resulte más fácil crear el fichero **makefile**. La **línea 1** contiene el ***destino simbólico* all**, y la ***lista con todos los destinos*** que queremos crear. En este caso sólo crearemos un destino, **bin/demo3**. A continuación tenemos que crear una regla para construir ese destino. En la **línea 3** aparece el ***destino* bin/demo3** y su ***lista de dependencias*** (ficheros que si se modifican obligan a crear de nuevo el destino). Como podemos ver en el gráfico, **bin/demo3** depende de tres módulos objeto, **obj/demo3.o**, **obj/adicion3.o** y **obj/producto3.o**. En la **línea 4** incluimos la orden ***g++*** que crea el ejecutable a partir de esos tres ficheros **.o**. Ahora tenemos que crear tres reglas para describir cómo se construyen cada uno de esos módulos objeto. En la **línea 6** se especifica que **obj/demo3.o** depende de **src/demo3.cpp** y además de **include/adicion3.h**, ya que la función ***main*** de **src/demo3.cpp** llama a las funciones ***suma*** y ***resta*** (pero no a ***multiplica*** y ***divide***, por lo que no depende de **include/producto3.h**). En la **línea 7** se incluye la orden ***g++*** correspondiente, añadiendo la opción **-I./include** para indicar en qué directorio se tiene que buscar el fichero de cabecera (**.h**). En la **línea 9** se añade una nueva regla que permite crear **obj/adicion3.o** a partir del **.cpp** y el **.h** correspondiente. La orden encargada de ello aparece en la **línea 10**. Finalmente, las **líneas 12 y 13** indican cómo generar **obj/producto3.o** a partir de su **.cpp** y su **.h**.
Comprobemos que el **makefile** que hemos creado funciona correctamente.

```
profesor@DECSAI:~/mp$ make -f make3
Creando el modulo objeto obj/demo3.o...
g++ -c -I./include -o obj/demo3.o src/demo3.cpp
Creando el modulo objeto obj/adicion3.o...
g++ -c -I./include -o obj/adicion3.o src/adicion3.cpp
Creando el modulo objeto obj/producto3.o...
g++ -c -I./include -o obj/producto3.o src/producto3.cpp
Creando el ejecutable bin/demo3...
g++ -o bin/demo3 obj/demo3.o obj/adicion3.o obj/producto3.o
Borrando los ficheros .o del directorio obj...
rm obj/*.o
profesor@DECSAI:~/mp$
```

A continuación vamos a modificar el fichero **makefile** anterior para crear una **biblioteca** con los módulos objeto **obj/adicion3.o** y **obj/producto3.o**, tal y como aparece en la siguiente figura. A esta **biblioteca** la vamos a llamar **liboper3.a** y la guardaremos en el directorio **lib**.



make3

```
1  all : bin/demo3 clean
2
3  bin/demo3 : obj/demo3.o lib/liboper3.a
4  »        @echo Creando el ejecutable bin/demo3...
5  »        g++ -o bin/demo3 obj/demo3.o -L./lib -loper3
6
7  obj/demo3.o : src/demo3.cpp include/adicion3.h
8  »        @echo Creando el modulo objeto obj/demo3.o...
9  »        g++ -c -I./include -o obj/demo3.o src/demo3.cpp
10
11 lib/liboper3.a : obj/adicion3.o obj/producto3.o
12 »        @echo Creando la biblioteca lib/liboper3.a...
13 »        ar rvs lib/liboper3.a obj/adicion3.o obj/producto3.o
14
15 obj/adicion3.o : src/adicion3.cpp include/adicion3.h
16 »        @echo Creando el modulo objeto obj/adicion3.o...
17 »        g++ -c -I./include -o obj/adicion3.o src/adicion3.cpp
18
19 obj/producto3.o : src/producto3.cpp include/producto3.h
20 »        @echo Creando el modulo objeto obj/producto3.o...
21 »        g++ -c -I./include -o obj/producto3.o src/producto3.cpp
22
23 clean :
24 »        @echo Borrando los ficheros .o del directorio obj...
25 »        rm obj/*.o
26
```

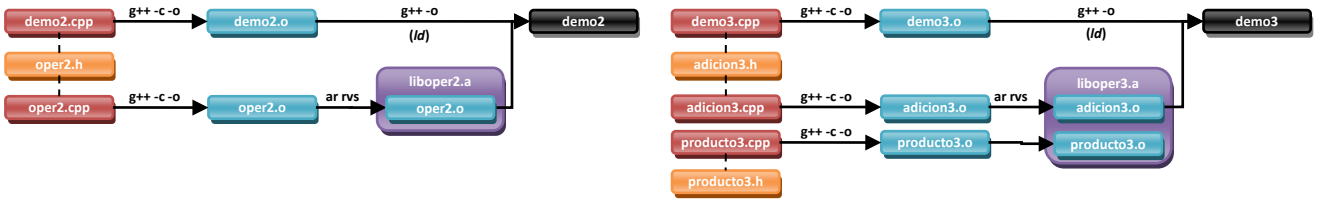
Como podemos ver en la **línea 3**, la única diferencia es que ahora **bin/demo3** depende de la **biblioteca lib/liboper3.a**, en lugar de los dos módulos objeto **obj/adicion3.o** y **obj/producto3.o**. En la **línea 4** la orden ***g++*** se modifica para enlazar la **biblioteca** con la opción **-loper3** e indicando el directorio en el que debe buscarse con la opción **-L./lib**. En la **línea 9** se añade una nueva regla que indica cómo construir ese destino **lib/liboper3.a** a partir de los dos módulos objeto. Recordemos que, tal y como aparece en la **línea 10**, las **bibliotecas** se crean utilizando la orden ***ar*** incluyendo las opciones **rvs**. El nombre de la **biblioteca** debe ser **lib/liboper3.a**, es decir, debe empezar por **lib**, para que luego podamos enlazarla escribiendo **-loper3**.

Ejecutemos el programa ***make*** para ver si se crea bien la **biblioteca**.

```
profesor@DECSAI:~/mp$ make -f make3
Creando el modulo objeto obj/demo3.o...
g++ -c -I./include -o obj/demo3.o src/demo3.cpp
Creando el modulo objeto obj/adicion3.o...
g++ -c -I./include -o obj/adicion3.o src/adicion3.cpp
Creando el modulo objeto obj/producto3.o...
g++ -c -I./include -o obj/producto3.o src/producto3.cpp
Creando la biblioteca lib/liboper3.a...
```

```
ar rvs lib/liboper3.a obj/adicion3.o obj/producto3.o
ar: creating lib/liboper3.a
a - obj/adicion3.o
a - obj/producto3.o
Creando el ejecutable bin/demo3...
g++ -o bin/demo3 obj/demo3.o -L./lib -loper3
Borrando los ficheros .o del directorio obj...
rm obj/*.o
profesor@DECSAI:~/mp$
```

Analicemos ahora qué diferencias hay entre los programas **demo2** y **demo3** creados a partir de las **bibliotecas** **liboper2.a** y **liboper3.a**. Podemos ver las dependencias de cada uno de ellos en el siguiente gráfico.



Recordemos que las funciones ***main*** de **demo2** y **demo3** solamente utilizaban las funciones ***suma*** y ***resta*** (***multiplica*** y ***divide*** no). En el primer caso, las cuatro funciones (***suma***, ***resta***, ***multiplica*** y ***divide***) se definían en un mismo fichero fuente llamado **oper2.cpp**. En el segundo, las funciones ***suma*** y ***resta*** se definían en **adicion3.cpp** mientras que ***multiplica*** y ***divide*** se definían en **producto3.cpp**. Así, la primera **biblioteca**, **liboper2.a**, contiene **un** solo módulo objeto (**oper2.o**), mientras que la segunda, **liboper3.a**, contiene **dos** (**adicion3.o** y **producto3.o**). Podemos listar el contenido del directorio **lib** para comparar el tamaño de ambas **bibliotecas**. Esto lo hacemos con la orden ***ls*** especificando la opción **-l** (menos 'ele') para que se muestre el tamaño de cada fichero, ya que en caso contrario solamente se mostrarán los nombres de los mismos.

```
profesor@DECSAI:~/mp$ ls -l lib
total 8
-rw-r--r-- 1 profesor profesor 1020 2009-03-23 14:39 liboper2.a
-rw-r--r-- 1 profesor profesor 1734 2009-03-23 14:47 liboper3.a
profesor@DECSAI:~/mp$
```

Como podemos apreciar, el tamaño de la **biblioteca** **liboper3.a** es mayor que el de **liboper2.a** ya que, aunque ambas contienen las mismas cuatro funciones, la primera **biblioteca** contiene dos módulos objeto (ficheros **.o**) en lugar de uno, por lo que el ***tamaño del índice*** de los módulos que la componen es mayor. Sin embargo, miremos ahora el tamaño de los ejecutables generados.

```
profesor@DECSAI:~/mp$ ls -l bin
total 72
-rwxr-xr-x 1 profesor profesor 9741 2009-03-23 14:18 ArgumentosMain
-rwxr-xr-x 1 profesor profesor 9924 2009-03-23 14:37 demo1
-rwxr-xr-x 1 profesor profesor 9990 2009-03-23 14:40 demo2
-rwxr-xr-x 1 profesor profesor 9932 2009-03-23 14:47 demo3
-rwxr-xr-x 1 profesor profesor 9689 2009-03-23 14:09 HolaMundo
-rwxr-xr-x 1 profesor profesor 9690 2009-03-23 14:14 HolaMundo2
profesor@DECSAI:~/mp$
```

Ahora ocurre justo lo contrario, el ejecutable **demo3** ocupa menos que **demo2**. La razón es que el **enlazador** solamente enlaza los módulos objeto de la **biblioteca** que ***realmente se van a usar*** (se enlazan ***módulos completos***, no funciones aisladas dentro de cada módulo). Así, en el primer caso (**demo2**) se incluye el código de las cuatro funciones aunque en realidad sólo llegamos a usar dos (***suma*** y ***resta***), ya que todas están incluidas en un mismo módulo objeto (**oper2.o**). Se está incluyendo código innecesario (las funciones ***multiplica*** y ***divide***) que aumenta el tamaño del ejecutable. En el segundo caso (**demo3**), solamente se enlaza el código de las funciones ***suma*** y ***resta*** (**adicion3.o**), ya que están en un módulo objeto diferente al de las funciones ***multiplica*** y ***divide*** (**producto3.o**), que no se utilizan. Por lo tanto, el tamaño del ejecutable **demo3** es algo menor. Aquí tenemos otra razón por la que debemos **modularizar el software**.

Uno de los problemas que podemos ver en los ficheros **makefile** que hemos escrito hasta ahora es que se repiten por todos lados los nombres de los directorios **bin**, **include**, **lib**, **obj** y **src**. ¿Qué pasa ahora si decidimos ***cambiar el nombre*** de alguno de ellos? Por ejemplo, ahora decidimos que los ejecutables se van a almacenar en un directorio llamado **exe**, los ficheros de cabecera en el directorio **h**, las bibliotecas en el directorio **a**, los módulos objetos en el directorio **o**, y los códigos fuente en el directorio **cpp**. Entonces tendríamos que ir cambiando ***una por una*** todas las ocurrencias de dichos directorios. De acuerdo, cualquier editor de textos por malo que sea incluye una opción de buscar y reemplazar que podemos utilizar para facilitarnos este proceso. Pero, ¿no sería mejor que pudiésemos definir el nombre de estos directorios ***una sola vez*** y que luego los pudiésemos utilizar todas las veces que quisiésemos? Así, si algún día decidimos cambiarlos, sólo tendremos que escribirlos una sola vez. Esto se consigue utilizando **macros**, que son ***variables o cadenas que se expanden*** cuando realizamos la llamada a ***make***. Es como una especie de ***#define*** pero sobre ficheros **makefile**. Por ejemplo, podemos tener al principio de nuestro fichero **makefile** una macro que sea **BIN = ./bin**, y luego siempre que queramos hacer referencia al directorio **./bin** escribiremos **$(BIN)**, de forma que después cada **$(BIN)** se sustituirá por **./bin**. De esta forma, si ahora quisiésemos cambiar el directorio en el que se guardan los ejecutables solamente tendríamos que modificar la **macro BIN = ./src**.

Veamos cómo quedaría el fichero **make3** utilizando **macros** para los nombres de los directorios.

make3

```
1  BIN = ./bin
2  INCLUDE = ./include
3  LIB = ./lib
4  OBJ = ./obj
5  SRC = ./src
6
7  all : $(BIN)/demo3 clean
8
9  $(BIN)/demo3 : $(OBJ)/demo3.o $(LIB)/liboper3.a
10 »        @echo Creando el ejecutable $(BIN)/demo3...
11 »        g++ -o $(BIN)/demo3 $(OBJ)/demo3.o -L$(LIB) -loper3
12
13 $(OBJ)/demo3.o : $(SRC)/demo3.cpp $(INCLUDE)/adicion3.h
14 »        @echo Creando el modulo objeto $(OBJ)/demo3.o...
15 »        g++ -c -I$(INCLUDE) -o $(OBJ)/demo3.o $(SRC)/demo3.cpp
16
17 $(LIB)/liboper3.a : $(OBJ)/adicion3.o $(OBJ)/producto3.o
18 »        @echo Creando la biblioteca $(LIB)/liboper3.a...
19 »        ar rvs $(LIB)/liboper3.a $(OBJ)/adicion3.o $(OBJ)/producto3.o
20
21 $(OBJ)/adicion3.o : $(SRC)/adicion3.cpp $(INCLUDE)/adicion3.h
22 »        @echo Creando el modulo objeto $(OBJ)/adicion3.o...
23 »        g++ -c -I$(INCLUDE) -o $(OBJ)/adicion3.o $(SRC)/adicion3.cpp
24
25 $(OBJ)/producto3.o : $(SRC)/producto3.cpp $(INCLUDE)/producto3.h
26 »        @echo Creando el modulo objeto $(OBJ)/producto3.o...
27 »        g++ -c -I$(INCLUDE) -o $(OBJ)/producto3.o $(SRC)/producto3.cpp
28
29 clean :
30 »        @echo Borrando los ficheros .o del directorio $(OBJ)...
31 »        rm $(OBJ)/*.o
32
```

Como podemos ver, las **macros** se sitúan ***una por línea***, normalmente al principio del fichero **makefile**. Hay que destacar que los nombres de las **macros** (lo que hay a la izquierda del igual) son ***sensibles a mayúsculas*** (en general se suelen escribir en mayúscula), y ***no pueden contener espacios en blanco***. Las **macros** no sólo se utilizan para ***representar listas de nombres de ficheros o directorios***, sino que también se pueden usar para otras muchas cosas como ***especificar opciones del compilador***, ***programas a ejecutar***, etc. De esta forma, es fácil crear ficheros **makefile** genéricos o plantillas que se adapten a diferentes proyectos software.

Aunque en este caso no resulta demasiado útil, se pueden ***sustituir caracteres*** en una **macro** previamente definida. Por ejemplo, la **macro $(INCLUDE:include=incDir)** sustituye, en **$(INCLUDE)**, el texto **include** por el texto **incDir**, con lo que se evaluaría como **./incDir** en lugar de **./include**.

También existen una serie de **macros** ***predefinidas***. Las más utilizadas son tres: **$^** equivale a ***todas las dependencias de la regla*** separadas entre sí por un espacio en blanco, **$<** equivale al nombre de la ***primera dependencia de la regla***, y **$@** equivale al nombre del ***destino de la regla***. Por ejemplo, modifiquemos el fichero **make3** para incluir una nueva regla cuyo destino llamaremos **backup**:

make3

```
1  BIN = ./bin
2  INCLUDE = ./include
3  LIB = ./lib
4  OBJ = ./obj
5  SRC = ./src
6
7  all : $(BIN)/demo3 backup clean
8
9  $(BIN)/demo3 : $(OBJ)/demo3.o  $(LIB)/liboper3.a
10 »         @echo Creando el ejecutable $(BIN)/demo3...
11 »         g++ -o $(BIN)/demo3 $(OBJ)/demo3.o -L$(LIB) -loper3
12
13 $(OBJ)/demo3.o : $(SRC)/demo3.cpp $(INCLUDE)/adicion3.h
14 »         @echo Creando el modulo objeto $(OBJ)/demo3.o...
15 »         g++ -c -I$(INCLUDE) -o $(OBJ)/demo3.o $(SRC)/demo3.cpp
16
17 $(LIB)/liboper3.a : $(OBJ)/adicion3.o $(OBJ)/producto3.o
18 »         @echo Creando la biblioteca $(LIB)/liboper3.a...
19 »         ar rvs $(LIB)/liboper3.a $(OBJ)/adicion3.o $(OBJ)/producto3.o
20
21 $(OBJ)/adicion3.o : $(SRC)/adicion3.cpp $(INCLUDE)/adicion3.h
22 »         @echo Creando el modulo objeto $(OBJ)/adicion3.o...
23 »         g++ -c -I$(INCLUDE) -o $(OBJ)/adicion3.o $(SRC)/adicion3.cpp
24
25 $(OBJ)/producto3.o : $(SRC)/producto3.cpp $(INCLUDE)/producto3.h
26 »         @echo Creando el modulo objeto $(OBJ)/producto3.o...
27 »         g++ -c -I$(INCLUDE) -o $(OBJ)/producto3.o $(SRC)/producto3.cpp
28
29 clean :
30 »         @echo Borrando los ficheros .o del directorio $(OBJ)...
31 »         rm $(OBJ)/*.o
32
33 backup : $(BIN)/demo3 $(LIB)/liboper3.a $(OBJ)/demo3.o $(OBJ)/adicion3.o $(OBJ)/producto3.o
   $(SRC)/demo3.cpp $(SRC)/adicion3.cpp $(SRC)/producto3.cpp $(INCLUDE)/adicion3.h
   $(INCLUDE)/producto3.h
34 »         @echo Creando el directorio para las copias de seguridad...
35 »         mkdir backup
36 »         @echo Realizando las copias de seguridad de los ejecutables...
37 »         cp $^ backup
38
```

Esta regla se encargará de realizar una ***copia de seguridad*** de todos los archivos (ejecutables, **bibliotecas**, módulos objeto, ficheros fuente y ficheros de cabecera) en el directorio **backup**. Si lanzamos el **makefile**, veremos que la **macro** ***predefinida*** **$^** de la **línea 37** se sustituye por ***todos los nombres de los ficheros de la lista de dependencias***, es decir, que en este caso la orden quedaría ***cp bin/demo3 lib/liboper3.a obj/demo3.o obj/adicion3.o obj/producto3.o src/demo3.cpp src/adicion3.cpp src/producto3.cpp include/adicion3.h include/producto3.h backup***.

```
profesor@DECSAI:~/mp$ make -f make3
Creando el modulo objeto ./obj/demo3.o...
g++ -c -I./include -o ./obj/demo3.o ./src/demo3.cpp
Creando el modulo objeto ./obj/adicion3.o...
g++ -c -I./include -o ./obj/adicion3.o ./src/adicion3.cpp
Creando el modulo objeto ./obj/producto3.o...
g++ -c -I./include -o ./obj/producto3.o ./src/producto3.cpp
Creando la biblioteca ./lib/liboper3.a...
ar rvs ./lib/liboper3.a ./obj/adicion3.o ./obj/producto3.o
r - ./obj/adicion3.o
r - ./obj/producto3.o
Creando el ejecutable ./bin/demo3...
g++ -o ./bin/demo3 ./obj/demo3.o -L./lib -loper3
Creando el directorio para las copias de seguridad...
mkdir backup
Realizando las copias de seguridad de los ejecutables...
```

```
cp bin/demo3 lib/liboper3.a obj/demo3.o obj/adicion3.o obj/producto3.o src/demo3.cpp src/adicion3.cpp
src/producto3.cpp include/adicion3.h include/producto3.h backup
Borrando los ficheros .o del directorio ./obj...
rm ./obj/*.o
profesor@DECSAI:~/mp$
```

También podríamos utilizar las **macros** ***predefinidas*** **$@** (***destino*** de la regla) y **$<** (***dependencia más a la izquierda*** de la lista de dependencias) para construir el resto de destinos. Por ejemplo, la regla que crea el ejecutable **demo3** quedaría de la siguiente manera:

```
 9 $(BIN)/demo3 : $(OBJ)/demo3.o  $(LIB)/liboper3.a
10 »        @echo Creando el ejecutable $@...
11 »        g++ -o $@ $< -L$(LIB) -loper3
```

donde **$@** representa el ***destino*** de la regla (**$(BIN)/demo3**) y **$<** la ***primera dependencia*** de la lista de dependencias (**$(OBJ)/demo3.o**).

Además de especificar los valores de las **macros** ***dentro del fichero*** **makefile**, podemos especificarlos ***en la propia llamada*** a ***make***. Para ver un ejemplo, vamos a modificar el fichero **make3** para incluir una nueva **macro** de nombre **DESTDIR**, a la cual no le vamos a asignar ningún valor dentro del **makefile**. Dicha macro se utilizará para especificar, en la llamada a ***make*** desde la línea de comandos, el nombre del directorio en el que queremos guardar el ejecutable.

make3

```
 1 INCLUDE = ./include
 2 LIB = ./lib
 3 OBJ = ./obj
 4 SRC = ./src
 5
 6 all : $(DESTDIR)/demo3 clean
 7
 8 $(DESTDIR)/demo3 : $(OBJ)/demo3.o $(LIB)/liboper3.a
 9 »        @echo Creando el ejecutable $(DESTDIR)/demo3...
10 »        mkdir $(DESTDIR)
11 »        g++ -o $(DESTDIR)/demo3 $(OBJ)/demo3.o -L$(LIB) –loper3
12
13 $(OBJ)/demo3.o : $(SRC)/demo3.cpp $(INCLUDE)/adicion3.h
14 »        @echo Creando el modulo objeto $(OBJ)/demo3.o...
15 »        g++ -c -I$(INCLUDE) -o $(OBJ)/demo3.o $(SRC)/demo3.cpp
16
17 $(LIB)/liboper3.a : $(OBJ)/adicion3.o $(OBJ)/producto3.o
18 »        @echo Creando la biblioteca $(LIB)/liboper3.a...
19 »        ar rvs $(LIB)/liboper3.a $(OBJ)/adicion3.o $(OBJ)/producto3.o
20
21 $(OBJ)/adicion3.o : $(SRC)/adicion3.cpp $(INCLUDE)/adicion3.h
22 »        @echo Creando el modulo objeto $(OBJ)/adicion3.o...
23 »        g++ -c -I$(INCLUDE) -o $(OBJ)/adicion3.o $(SRC)/adicion3.cpp
24
25 $(OBJ)/producto3.o : $(SRC)/producto3.cpp $(INCLUDE)/producto3.h
26 »        @echo Creando el modulo objeto $(OBJ)/producto3.o...
27 »        g++ -c -I$(INCLUDE) -o $(OBJ)/producto3.o $(SRC)/producto3.cpp
28
29 clean :
30 »        @echo Borrando los ficheros .o del directorio $(OBJ)...
31 »        rm $(OBJ)/*.o
32
```

Ahora llamamos a ***make*** especificando el valor de la **macro DESTDIR**.

```
profesor@DECSAI:~/mp$ make DESTDIR=/home/profesor/mp/bin -f make3
Creando el modulo objeto ./obj/demo3.o...
g++ -c -I./include -o ./obj/demo3.o ./src/demo3.cpp
Creando el modulo objeto ./obj/adicion3.o...
g++ -c -I./include -o ./obj/adicion3.o ./src/adicion3.cpp
Creando el modulo objeto ./obj/producto3.o...
g++ -c -I./include -o ./obj/producto3.o ./src/producto3.cpp
Creando la biblioteca ./lib/liboper3.a...
```

```
ar rvs ./lib/liboper3.a ./obj/adicion3.o ./obj/producto3.o
r - ./obj/adicion3.o
r - ./obj/producto3.o
Creando el ejecutable /home/profesor/mp/bin/demo3...
mkdir /home/profesor/mp/bin
mkdir: no se puede crear el directorio «/home/profesor/mp/bin»: El fichero ya existe
make: *** [/home/profesor/mp/bin/demo3] Error 1
profesor@DECSAI:~/mp$
```

Sin embargo, si no especificamos el valor de **DESTDIR** en la llamada, ***make*** mostrará un mensaje de error (***número de argumentos insuficiente***) y terminará de procesar el fichero **makefile** sin generar el ejecutable. También aparecerá un mensaje de error si ***el directorio ya existe***, ya que no se puede volver a crear. ¿Cómo se pueden solucionar estos problemas? Por un lado podemos utilizar el prefijo **-** (signo menos) antes de la orden **mkdir** para evitar que ***make*** deje de procesar el fichero **makefile** cuando esta instrucción produce algún error, ya sea porque no se le proporciona el nombre del directorio o porque este ya existe previamente. Aun así, si no se especifica el valor de **DESTDIR**, el ejecutable se guarda en **/demo3**, es decir, colgando del directorio raíz (**/**). Puede ocurrir que no tengamos permiso para escribir en ese directorio, por lo que no podrá guardarse el ejecutable. La solución es incluir en el fichero **makefile** una **macro** de nombre **BASE** que indique un ***directorio base*** en el que se guardará el ejecutable si no se asigna un valor a la **macro DESTDIR**.

make3

```
1  BASE = /home/profesor/mp/
2  INCLUDE = ./include
3  LIB = ./lib
4  OBJ = ./obj
5  SRC = ./src
6
7  all : $(BASE)$(DESTDIR)/demo3 clean
8
9  $(BASE)$(DESTDIR)/demo3 : $(OBJ)/demo3.o $(LIB)/liboper3.a
10 »        @echo Creando el ejecutable $(BASE)$(DESTDIR)/demo3...
11 »        -mkdir $(BASE)$(DESTDIR)
12 »        g++ -o $(BASE)$(DESTDIR)/demo3 $(OBJ)/demo3.o -L$(LIB) -loper3
13
14 $(OBJ)/demo3.o : $(SRC)/demo3.cpp $(INCLUDE)/adicion3.h
15 »        @echo Creando el modulo objeto $(OBJ)/demo3.o...
16 »        g++ -c -I$(INCLUDE) -o $(OBJ)/demo3.o $(SRC)/demo3.cpp
17
18 $(LIB)/liboper3.a : $(OBJ)/adicion3.o $(OBJ)/producto3.o
19 »        @echo Creando la biblioteca $(LIB)/liboper3.a...
20 »        ar rvs $(LIB)/liboper3.a $(OBJ)/adicion3.o $(OBJ)/producto3.o
21
22 $(OBJ)/adicion3.o : $(SRC)/adicion3.cpp $(INCLUDE)/adicion3.h
23 »        @echo Creando el modulo objeto $(OBJ)/adicion3.o...
24 »        g++ -c -I$(INCLUDE) -o $(OBJ)/adicion3.o $(SRC)/adicion3.cpp
25
26 $(OBJ)/producto3.o : $(SRC)/producto3.cpp $(INCLUDE)/producto3.h
27 »        @echo Creando el modulo objeto $(OBJ)/producto3.o...
28 »        g++ -c -I$(INCLUDE) -o $(OBJ)/producto3.o $(SRC)/producto3.cpp
29
30 clean :
31 »        @echo Borrando los ficheros .o del directorio $(OBJ)...
32 »        rm $(OBJ)/*.o
33
```

De esta forma, si realizamos la llamada a ***make*** sin especificar el valor de la **macro DESTDIR**, el ejecutable **demo3** se guardará en el directorio especificado por la macro **BASE**, **/home/profesor/mp**.

```
profesor@DECSAI:~/mp$ make -f make3
Creando el ejecutable /home/profesor/mp//demo3...
mkdir /home/profesor/mp/
mkdir: no se puede crear el directorio «/home/profesor/mp/»: El fichero ya existe
make: [/home/profesor/mp//demo3] Error 1 (no tiene efecto)
g++ -o /home/profesor/mp//demo3 ./obj/demo3.o -L./lib -loper3
Borrando los ficheros .o del directorio ./obj...
rm ./obj/*.o
profesor@DECSAI:~/mp$
```

Otra forma de solucionar este problema es utilizando ***directivas condicionales*** (muy parecidas a las del **preprocesador** de C++) que permiten a ***make dirigir el flujo de procesamiento*** a un bloque u otro dependiendo del resultado de la evaluación de una condición.

make3

```
1  INCLUDE = ./include
2  LIB = ./lib
3  OBJ = ./obj
4  SRC = ./src
5
6  all : $(DESTDIR)/demo3 clean
7
8  $(DESTDIR)/demo3 : $(OBJ)/demo3.o $(LIB)/liboper3.a
9  ifndef DESTDIR
10 »        @echo Error: Falta especificar opcion DESTDIR=directorio
11 else
12 »        @echo Creando el ejecutable $(DESTDIR)/demo3...
13 »        -mkdir $(DESTDIR)
14 »        g++ -o $(DESTDIR)/demo3 $(OBJ)/demo3.o -L$(LIB) -loper3
15 endif
16
17 $(OBJ)/demo3.o : $(SRC)/demo3.cpp $(INCLUDE)/adicion3.h
18 »        @echo Creando el modulo objeto $(OBJ)/demo3.o...
19 »        g++ -c -I$(INCLUDE) -o $(OBJ)/demo3.o $(SRC)/demo3.cpp
20
21 $(LIB)/liboper3.a : $(OBJ)/adicion3.o $(OBJ)/producto3.o
22 »        @echo Creando la biblioteca $(LIB)/liboper3.a...
23 »        ar rvs $(LIB)/liboper3.a $(OBJ)/adicion3.o $(OBJ)/producto3.o
24
25 $(OBJ)/adicion3.o : $(SRC)/adicion3.cpp $(INCLUDE)/adicion3.h
26 »        @echo Creando el modulo objeto $(OBJ)/adicion3.o...
27 »        g++ -c -I$(INCLUDE) -o $(OBJ)/adicion3.o $(SRC)/adicion3.cpp
28
29 $(OBJ)/producto3.o : $(SRC)/producto3.cpp $(INCLUDE)/producto3.h
30 »        @echo Creando el modulo objeto $(OBJ)/producto3.o...
31 »        g++ -c -I$(INCLUDE) -o $(OBJ)/producto3.o $(SRC)/producto3.cpp
32
33 clean :
34 »        @echo Borrando los ficheros .o del directorio $(OBJ)...
35 »        rm $(OBJ)/*.o
36
```

Así, una vez que se han generado todos los módulos objeto y la biblioteca **liboper3.a**, cuando se intenta crear el ejecutable **demo3** con la regla de la **línea 8**, si la **macro DESTDIR** no está definida, es decir, si no se ha especificado un directorio de destino en la llamada a ***make***, se muestra un mensaje de error por pantalla y no se crea el ejecutable. Por el contrario, si la **macro** está definida, ***make*** ejecuta las tareas de la parte **else** (**líneas 12, 13 y 14**), intentando crear el directorio de destino y generando el ejecutable.

Veamos que efectivamente se muestra un mensaje de error si no se define la **macro DESTDIR** en la llamada a ***make***.

```
profesor@DECSAI:~/mp$ make -f make3
Creando el modulo objeto ./obj/demo3.o...
g++ -c -I./include -o ./obj/demo3.o ./src/demo3.cpp
Creando el modulo objeto ./obj/adicion3.o...
g++ -c -I./include -o ./obj/adicion3.o ./src/adicion3.cpp
Creando el modulo objeto ./obj/producto3.o...
g++ -c -I./include -o ./obj/producto3.o ./src/producto3.cpp
Creando la biblioteca ./lib/liboper3.a...
ar rvs ./lib/liboper3.a ./obj/adicion3.o ./obj/producto3.o
r - ./obj/adicion3.o
r - ./obj/producto3.o
Error: Falta especificar opcion DESTDIR=directorio
Borrando los ficheros .o del directorio ./obj...
rm ./obj/*.o
profesor@DECSAI:~/mp$
```

Uno de los defectos que le podemos ver a **make3** es que ***hay reglas que se parecen mucho***. Por ejemplo, las reglas que generan los módulos objeto **adicion3.o** y **producto3.o** son prácticamente idénticas. Si en la primera regla cambiamos todas las ocurrencias de la palabra **adicion** por la palabra **producto**, obtenemos la segunda regla. Entonces, si las reglas son tan iguales, ¿hace falta escribirlas todas? En realidad no es necesario. ***make*** utilizará ***reglas implícitas*** (que no hay que escribir) para construir aquellos destinos para los que no hayamos especificado ninguna ***regla explícita***. Existe por ejemplo una ***regla implícita*** que indica cómo obtener el fichero objeto (**.o**) a partir del fichero fuente (**.cpp**):

```
%.o : %.cpp
»       $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@
```

En realidad esta es una ***regla implícita patrón***. La diferencia con las ***reglas normales*** es que las ***reglas patrón*** contienen en el destino el carácter **%** en alguna parte (y una sola vez). En este caso, el destino **%.o** empareja con todos los ficheros con extensión **.o** (el carácter **%** es como un ***** o un **comodín**). Por tanto, esta ***regla patrón*** dice cómo construir cualquier fichero **.o** a partir del **.cpp** correspondiente.

Así, nosotros solamente tendríamos que definir los valores para las macros **CXX**, **CPPFLAGS** y **CXXFLAGS**, y ya sería ***make*** quién utilizaría esa ***regla implícita*** para construir los módulos objeto que no tuviesen asociada ninguna ***regla explícita*** en el fichero **makefile**. Es importante destacar este aspecto, ya que las ***reglas implícitas patrón*** solamente se aplican sobre destinos que emparejan con su patrón, siempre y cuando se encuentren las dependencias y ***no existan otras reglas*** que construyan dicho destino. Si hay más de una ***regla implícita*** que pueda aplicarse, sólo se aplicará una de ellas (ya depende del orden de las mismas).

Vamos a crear un nuevo fichero **makefile** llamado **make4** para practicar con las ***reglas implícitas***.

```
profesor@DECSAI:~/mp$ kwrite make4&
[19] 8304
profesor@DECSAI:~/mp$
```

make4

```
1  CXX = g++
2  CPPFLAGS =
3  CXXFLAGS = -I$(INCLUDE)
4  INCLUDE = ./include
5
6  all : demo3 clean
7
8  demo3 : demo3.o adicion3.o
9  »         g++ -o demo3 demo3.o adicion3.o
10
11 clean :
12 »         @echo Borrando los ficheros .o...
13 »         rm *.o
14
```

Para poder trabajar con estos ejemplos copiaremos los ficheros **src/demo3.cpp**, **src/adicion3.cpp**, e **include/adicion3.h** en el directorio base **mp**.

```
profesor@DECSAI:~/mp$ cp src/demo3.cpp .
profesor@DECSAI:~/mp$ cp src/adicion3.cpp .
profesor@DECSAI:~/mp$ cp include/adicion3.h .
profesor@DECSAI:~/mp$
```

En las **líneas 8 y 9** del fichero **make4** aparece la primera regla, que es una ***regla explícita*** que tiene asociada una ***orden*** que dice cómo crear el ejecutable **demo3** a partir de los dos módulos objeto, **demo3.o** y **adicion3.o**. Sin embargo, no existen ***reglas explícitas*** que digan cómo construir estos módulos objeto a partir de los ficheros fuente correspondientes. Como hemos visto, ***los módulos objeto dependen de forma implícita de los módulos fuente asociados***, por lo que no es necesario indicar esa dependencia para que el fichero **makefile** funcione.

```
profesor@DECSAI:~/mp$ make -f make4
g++ -I./include   -c -o demo3.o demo3.cpp
g++ -I./include   -c -o adicion3.o adicion3.cpp
g++ -o demo3 demo3.o adicion3.o
Borrando los ficheros .o...
rm *.o
profesor@DECSAI:~/mp$
```

En nuestro caso, aunque no especifiquemos la orden que crea los ficheros objeto, sí deberíamos especificar la ***lista de dependencias*** que tiene cada uno (***incluyendo los ficheros de cabecera***). De esta forma obligamos a que se reconstruyan **demo3.o** y **adicion3.o** cada vez que se modifica el fichero de cabecera **adicion3.h**.

make4

```
1  CXX = g++
2  CPPFLAGS =
3  CXXFLAGS = -I$(INCLUDE)
4  INCLUDE = ./include
5
6  all : demo3 clean
7
8  demo3 : demo3.o adicion3.o
9  »       g++ -o demo3 demo3.o adicion3.o
10
11 demo3.o : $(INCLUDE)/adicion3.h
12
13 adicion3.o : $(INCLUDE)/adicion3.h
14
15 clean :
16 »       @echo Borrando los ficheros .o...
17 »       rm *.o
18
```

Por esta razón es necesario añadir las **líneas 11 y 13**. Debemos remarcar que no estamos incluyendo ninguna orden para generar **demo3.o** y **adicion3.o**, sino que ***make*** utilizará una ***regla implícita*** para generar cada uno de ellos. Si realizamos la llamada a ***make***, podemos ver las órdenes que ejecuta.

```
profesor@DECSAI:~/mp$ make -f make4
g++ -I./include   -c -o demo3.o demo3.cpp
g++ -I./include   -c -o adicion3.o adicion3.cpp
g++ -o demo3 demo3.o adicion3.o
Borrando los ficheros .o...
rm *.o
profesor@DECSAI:~/mp$
```

Además de la ***regla implícita*** que se encarga de generar los ficheros objeto a partir de los ficheros fuente correspondientes, existe otra ***regla implícita*** para ***enlazar el programa***:

```
»       $(CC) $(LDFLAGS) $(LOADLIBS)
```

Esta regla funciona para programas con ***un solo fichero fuente***. Si tenemos más de un fichero objeto, uno de ellos debe tener ***el mismo nombre que el ejecutable*** para que funcione correctamente. Así, dando valores a las **macros CC**, **LDFLAGS** y **LOADLIBS** ya no haría falta ni siquiera especificar una orden para construir el ejecutable a partir de los módulos objeto.

make4

```
CXX = g++
CC = $(CXX)
CPPFLAGS =
CXXFLAGS = -I$(INCLUDE)
LDFLAGS =
LOADLIBS =
INCLUDE = ./include

all : demo3 clean

demo3 : demo3.o adicion3.o

demo3.o : $(INCLUDE)/adicion3.h

adicion3.o : $(INCLUDE)/adicion3.h

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

```

Podemos comprobar que las órdenes ejecutadas por ***make*** son las mismas.

```
profesor@DECSAI:~/mp$ make -f make4
g++ -I./include   -c -o demo3.o demo3.cpp
g++ -I./include   -c -o adicion3.o adicion3.cpp
g++   demo3.o adicion3.o   -o demo3
Borrando los ficheros .o...
rm *.o
profesor@DECSAI:~/mp$
```

También podemos utilizar las ***reglas implícitas patrón*** para definir nuestras propias ***reglas implícitas*** o incluso adaptar el comportamiento de las que ya hay definidas. Para ello, las escribimos en nuestro fichero **makefile** con el mismo destino y lista de dependencias, pero ***modificando la orden asociada***, pudiendo incluso no dejar ninguna orden si queremos que ***make*** ignore esa ***regla implícita***.

```
%.o : %.cpp %.h
	g++ -c $< -o $@
```

Otras reglas muy parecidas a las ***reglas implícitas patrón*** son las ***reglas patrón estáticas***. La diferencia es que las ***reglas implícitas*** se intentan aplicar ***a todos los destinos*** que emparejan con el patrón destino, mientras que las ***reglas patrón estáticas*** especifican ***una lista de destinos*** sobre los que se aplicará la regla.

```
OBJETOS = demo3.o adicion3.o

$(OBJETOS) : %.o %.cpp
	g++ -c $(CFLAGS) $< -o $@
```

Así, la regla anterior solamente se aplicaría a los ficheros **demo3.cpp** y **adicion3.cpp**, y no a cualquier otro fichero con extensión **.cpp**.

Para finalizar, comentamos algunas opciones del compilador ***g++*** que pueden resultarnos útiles en el futuro: **-g** incluye en el ejecutable la información necesaria para poder trazarlo usando un ***depurador***, **-v** muestra con ***detalle*** en **stderr** las órdenes ejecutadas por ***g++***, **-O**, **-O1**, **-O2**, **-O3** y **-Os** ***optimizan*** el tamaño y la velocidad del código compilado (así podemos reducir automáticamente el tamaño de nuestros ejecutables y hacer que estos se ejecuten más rápido), y **-Wall** muestra todos los mensajes de ***advertencia*** (***warning***) del compilador.