Metodología de la programación

14 de febrero de 2020

ugr | Universidad de Granada

Práctica 1: modularización y compilación separada

Curso 2019-2020

Índice

Objetivos

El objetivo de esta práctica es trabajar los conceptos básicos de modularización (organización del código en varios ficheros diferentes) y compilación separada (generación del ejecutable final a partir de todos los ficheros usados).

Punto de partida

Para partir de código ya disponible usaremos dos clase diferentes, cuyo esqueleto está disponible en la web de la asignatura en **PRADO**. La clase **Punto** define dos datos miembro de tipo privado. La clase **Circulo** define también dos datos miembro privados: uno de tipo **Punto** para identificar su centro y otro de tipo **double** para indicar el valor del radio.

Para centrarnos en la modularización del código y en su compilación separada, partimos de un archivo con el esqueleto inicial del código a desarrollar: **circuloMedio.cpp**. Presenta la estructura de las dos clases indicadas previamente, incluyendo la función **main** con todas las sentencias necesarias para comprobar que el código funciona de forma correcta. Los métodos a implementar están marcados por una línea de puntos:

- clase **Punto**
  - constructor por defecto
  - constructor recibiendo como argumentos sus coordenadas
  - método para devolver el valor del dato miembro **x**: **getX**
  - método para devolver el valor del dato miembro **y**: **getY**
  - método para asignar el valor del dato miembro **x**: **setX**
  - método para asignar el valor del dato miembro **y**: **setY**
  - método para mostrar el valor de los datos miembro: **mostrar**. Método a implementar fuera del cuerpo de la clase

    - método para leer los valores de los datos miembro: **leerDatos**. Método a implementar fuera del cuerpo de la clase

- clase **Circulo**
    - constructor por defecto
    - constructor recibiendo como argumento el valor de los datos miembro (**vcentro**, objeto de la clase **Punto** y **vradio**)
    - método para devolver el valor del dato miembro **centro**: **getCentro**
    - método para devolver el valor del dato miembro **radio**: **getRadio**
    - método para asignar el valor del dato miembro **centro**: **setCentro**
    - método para asignar el valor del dato miembro **radio**: **setRadio**
    - método para mostrar el contenido del objeto: **mostrar**. Este método se implementará fuera del cuerpo de la clase
    - método leer por teclado los valores de los datos miembro: **leerDatos**. A implementar fuera del cuerpo de la clase
    - método para calcular el área de un círculo: **calcularArea**. Ya está implementado fuera del cuerpo de la clase

Existen además una serie de funciones no vinculadas a las clases anteriores, por lo que no forman parte de ninguna de ellas:

- **calcularDistancia**, recibiendo como argumento dos objetos de la clase **Punto**
- **calcularPuntoMedio**, recibiendo como argumento dos objetos de la clase **Punto**
- **calcularDistancia**, con dos objetos de tipo **Circulo** como argumento. La función calcula la distancia entre los centros de los círculos. Ya está implementado
- **comprobarInterior**, con objeto de la clase **Punto** y **Circulo** como argumento. Determina si el punto pasado como primer argumento es interior al círculo pasado como segundo argumento. Ya está implementado.

El cuerpo de la clase sólo debería contener las declaraciones de los métodos y la implementación de aquellos que sean muy elementales (una o dos líneas de código). El resto de métodos deben implementarse fuera del cuerpo de la clase.

Al final del archivo se incluye la función **main** para probar la funcionalidad implementada. Para ello se puede usar el siguiente caso de prueba:

- círculo 1: radio 3 y centro $(0, 0)$
- círculo 2: radio 4 y centro $(4, 0)$
- con estos datos debe obtenerse que el círculo calculado, que pasa por los centros de los círculos anteriores debe tener como radio 2, siendo su centro $(2, 0)$

Paso 1: un único archivo

El primer paso consistirá en compilar el código anterior mediante la siguiente sentencia (en la línea de comandos):

```
g++ -o circuloMedio circuloMedio.cpp
```

La opción **-o** permite especificar cómo se llamará el ejecutable a generar. Para ejecutar el programa, si no hubo problemas de compilación, haremos:

```
./circuloMedio
```

Paso 2: organización en módulos

Se trata ahora de obtener una nueva versión del programa procediendo a dividir el código en módulos diferentes. Se recomienda trabajar en una carpeta diferente, llamada **paso2**. Los módulos a crear son:

- **Punto**: implementado en los archivos **punto.h** y **punto.cpp**. Contiene todo el código relativo a la clase del mismo nombre
- **Circulo**: archivos **circulo.h** y **circulo.cpp**. Desde este módulo se hace uso del módulo anterior
- **Utilidades**: archivo **utilidades.cpp** y **utilidades.h**. Se incluyen aquí las funciones de utilidad que no están directamente relacionadas con las clases anteriores. Se precisará el uso de los módulos de las clases **Punto** y **Circulo**. El archivo con extensión **h** contendrá únicamente la declaración de las funciones
- **Principal**: implementado en el archivo **principal.cpp**. Se incluye aquí el programa principal (**main**) y se precisa el uso de los dos módulos anteriores

Para evitar errores en la inclusión múltiple de declaraciones conviene usar las siguientes directivas de preprocesamiento en los archivos con extensión **.h** (se incluye, como ejemplo, las directivas a incluir para la clase **Punto** en el archivo **punto.h**):

```
#ifndef PUNTO_H
#define PUNTO_H
 ......................
#endif
```

Como ya se ha indicado, el reparto de declaraciones y métodos es el siguiente:

- archivo **punto.h**: incluye la declaración de la clase **Punto**. Los métodos sencillos, como constructores y métodos de acceso se implementan directamente en la declaración de la clase. En concreto: constructor por defecto, constructor con argumentos, **getX**, **getY**, **setX**, **setY**. La cabecera contiene las líneas con las directivas de compilación condicional indicadas arriba.
- archivo **punto.cpp**: contiene la implementación de los métodos **mostrar** y **leerDatos**. Incluirá el archivo de declaración de la clase **Punto**. Son también necesarias las directivas **include** para permitir la funcionalidad de los métodos de la clase (como hay métodos que contienen sentencias de entrada y salida tendremos que incluir la librería **iostream** y el uso del espacio de nombres estándar). En definitiva, la cabecera del archivo contendrá las siguientes líneas:

```
#include<iostream>
#include "punto.h"

using namespace std;
.....................
```

- archivo **circulo.h**: incluye la declaración de la clase **Circulo**. Igual que en el caso de la clase **Punto**, también se proporcionan en el cuerpo de la clase las implementaciones de los constructores y métodos de acceso sencillos. Debe incluirse el archivo de cabecera de la clase **Punto**. Los métodos implementados aquí serán: constructor por defecto, constructor con argumentos, **getCentro**, **getRadio**, **setCentro** y **setRadio**. El archivo de cabecera contiene las correspondientes líneas con directivas de compilación condicional:

```
#ifndef CIRCULO_H
#define CIRCULO_H

#include "punto.h"
.....................
#endif
```

- archivo **circulo.cpp**: contiene la implementación de los métodos **mostrar**, **leerDatos** y **calcularArea**. Incluirá su correspondiente archivo de cabecera (**circulo.h**) y precisa además la inclusión de **iostream** y **cmath**. También debe indicar el uso del espacio de nombres **std**. Por esta razón, la cabecera del archivo contendrá:

```
#include<iostream>
#include <cmath>
#include "circulo.h"

using namespace std;
.....................
```

- archivo **utilidades.h**: contiene la declaración (sólo la declaración) de los métodos del módulo: cálculo de distancia entre dos objetos de la clase **Punto**, cálculo del punto medio entre dos puntos, cálculo de distancia entre dos círculos y método de comprobación de pertenencia de un punto a un círculo. Es decir, la declaración de **calcularDistancia(Punto, Punto)**, **calcularPuntoMedio(Punto, Punto)**, **calcularDistancia(Circulo, Circulo)** y **comprobarInterior(Punto, Circulo)**. Precisa la inclusión de **punto.h** y **circulo.h**. Las directivas de compilación condicional a incluir son:

```
#ifndef UTILIDADES_H
#define UTILIDADES_H

#include "punto.h"
#include "circulo.h"
.....................
#endif
```

- archivo **utilidades.cpp**: contiene la implementación de los métodos del módulo, junto con la directivas **include** de su archivo de cabecera y de **cmath**:

```
#include <cmath>
#include "utilidades.h"

.....................
```

- archivo **principal.cpp**: contiene el programa principal que permite comprobar el funcionamiento del código implementado. Debe presentar directivas **include** para **iostream**, **punto.h**, **circulo.h** y **utilidades.h**

NOTA: conviene tener en cuenta que nunca deberíamos usar la sentencia **using namespace std** en los archivos de cabecera, ya que podría terminar generando problemas. De esta forma, esta sentencia se incluirá únicamente en archivos con extensión **cpp**. La cabecera del archivo debe contener:

```
#include<iostream>
#include "punto.h"
#include "circulo.h"
#include "utilidades.h"

using namespace std;
.....................
```

Una vez organizado el código de esta forma tendremos que proceder a su compilación, con la siguiente secuencia de órdenes, que generan un ejecutable llamado **principal**:

```
g++ -c -o punto.o punto.cpp
g++ -c -o circulo.o circulo.cpp
g++ -c -o utilidades.o utilidades.cpp
g++ -c -o principal.o principal.cpp

g++ -o principal punto.o circulo.o utilidades.o principal.o
```

Podemos plantearnos ahora qué necesitamos hacer para volver a generar el ejecutable tras un cambio en alguno de los archivos disponibles: ¿a la fuerza tenemos que volver a compilar todo? Ya veremos más adelante la forma de abordar este problema.

Paso 3: creación de biblioteca

Trabajaremos ahora en un directorio nuevo llamado **paso3**. Copiamos al directorio nuevo los archivos de código fuente y cabecera del paso anterior. A partir de ellos vamos a describir la secuencia de pasos para generar una biblioteca que podría utilizarse en futuros programas. Se siguen los siguientes pasos:

- compilación de los archivos **cpp**

```
g++ -c -o punto.o punto.cpp
g++ -c -o circulo.o circulo.cpp
g++ -c -o utilidades.o utilidades.cpp
g++ -c -o principal.o principal.cpp
```

- se crea una biblioteca con los archivos **punto.o**, **circulo.o** y **utilidades.o** a la que llamaremos **libformas.a**:

```
ar rvs libformas.a punto.o circulo.o utilidades.o
```

- ahora se crea el ejecutable a partir de la biblioteca (y no de los ficheros objeto):

```
g++ -L. -o principal principal.o -lformas
```

En el paso final se observa que:

- la opción **-L.** permite indicar que la ruta de búsqueda de librerías se reduce al directorio actual (desde el que se ejecuta la orden, representado por un punto). También cabe destacar que la mención de la librería a incluir se hace mediante la opción **-l**, seguida por el nombre de la librería, pero omitiendo el prefijo **lib**.

Paso 4: organización de los archivos en carpetas

En este paso vamos a trabajar en el directorio **paso4** y procederemos a agrupar los diferentes recursos del programa: archivos de código fuente, archivos de cabecera, archivos objeto resultado de la compilación, biblitecas y binario final (ejecutable). Estos directorios se llamarán **src**, **include**, **obj**, **lib** y **bin**, respectivamente. Ahora la generación del código final precisa seguir los siguientes pasos:

- copiar los archivos con extensión **cpp** al directorio **src**
- copiar los archivos con extensión **h** al directorio **include**
- compilación de los archivos de código fuente. Para ello nos situamos en el directorio del paso 4 e indicamos al compilador la ruta en que localizar los archivos de cabecera y dónde dejar los ficheros objeto generados:

```
g++ -c -o obj/punto.o -Iinclude src/punto.cpp
g++ -c -o obj/circulo.o -Iinclude src/circulo.cpp
g++ -c -o obj/utilidades.o -Iinclude src/utilidades.cpp
g++ -c -o obj/principal.o -Iinclude src/principal.cpp
```

- se genera la biblioteca en el directorio **lib**:

```
ar rvs lib/libformas.a obj/punto.o obj/circulo.o obj/utilidades.o
```

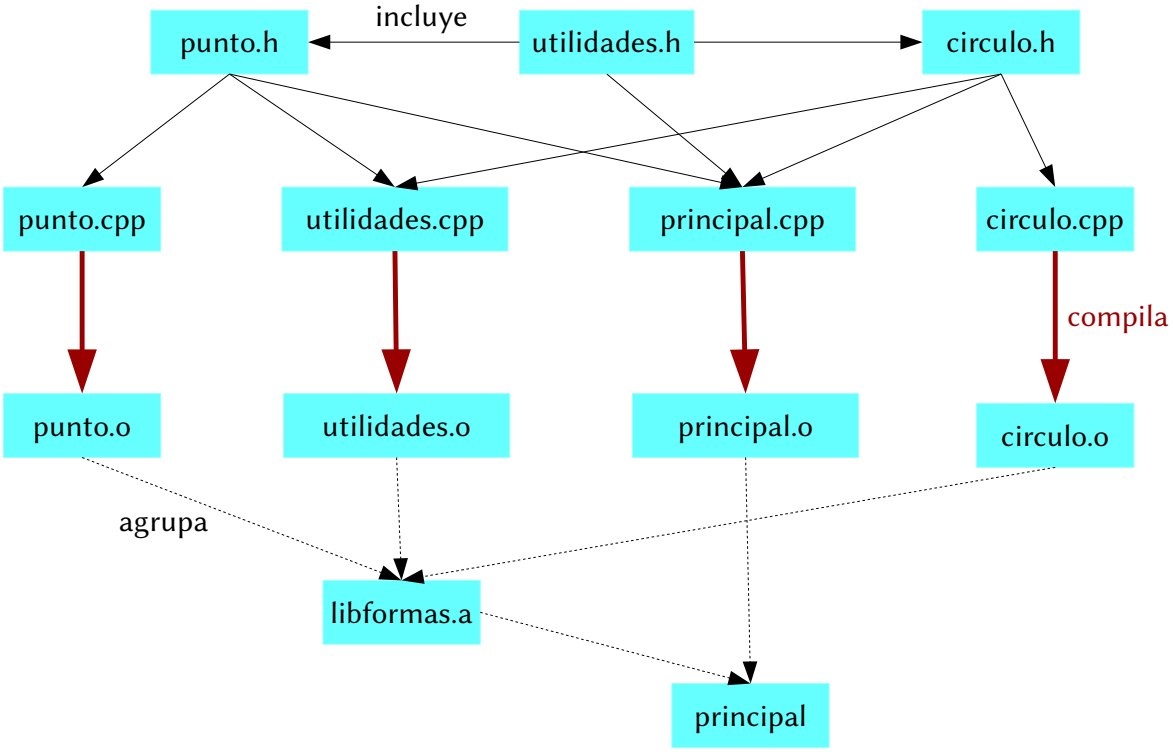
- construcción del ejecutable:

```
g++ -Llib -o bin/principal obj/principal.o -lformas
```

Paso 5: automatización del proceso de compilación

Se trabaja en un nuevo directorio (**paso5**), donde copiaremos los directorios generados en el punto anterior. Se trata ahora de construir un fichero, llamado **Makefile**, gracias al que la herramienta **make** puede gestionar de forma automática la compilación de nuestro código, haciendo en cada paso únicamente las operaciones necesarias. Recordad que en los puntos anteriores preguntábamos qué ocurre si se modifica algún archivo. Está claro que eso obligará a volver a generar el ejecutable, pero, ¿hay que repetir todas las operaciones?, ¿es posible ahorrar alguna?

Gracias al procedimiento que veremos ahora podemos olvidarnos de estos detalles: de forma automática se detectarán los cambios y se realizarán únicamente las operaciones de generación que resulten imprescindibles. El archivo **Makefile** se genera con un editor de texto plano y lo ubicamos en el directorio raíz de trabajo (en **paso5**). Antes de comenzar a trabajar en él debemos ser capaces de identificar de forma clara las dependencias que existen entre los archivos involucrados en la generación del código. Estas relaciones se muestran de forma esquemática en el siguiente gráfico:



Se aprecian en el gráfico tres tipos diferentes de relaciones: **incluye**, **compila** y **agrupa**. Las esenciales son las primeras, que representan las sentencias **include** que afectan a nuestros módulos. El hecho de fijar de forma correcta las dependencias hace que la generación del ejecutable sea eficiente, sin repetir operaciones innecesarias. Por ejemplo, un cambio en la clase **Círculo** no afecta a las operaciones de compilación sobre la clase **Punto**. Sin embargo, sí que implica la necesidad de modificar la biblioteca y, por supuesto, el programa principal. El objetivo de la herramienta **make** consiste en evitar que tengamos que estar recordando estas dependencias.

La herramienta **make** lee las instrucciones del archivo **Makefile** del proyecto y realiza todo el trabajo de forma automática. Este fichero recoge una serie de reglas necesarias para obtener el resultado final. El aspecto de una regla es el siguiente:

```
objetivo: lista de dependencias
  acciones a realizar
```

En cada regla:

- el objetivo indica qué se desea construir. Habitualmente es el nombre de un archivo a obtener como resultado del procesamiento del código (aunque no tiene por qué ser siempre así)
- la lista de dependencias indica de qué elementos depende la construcción del objetivo de la regla. Habitualmente se tratará de objetivos previos. La herramienta **make** comprobará que todas las dependencias están disponibles antes de poder ejecutar la regla (y si es necesario, hará lo posible por generar los elementos necesarios, siempre que disponga de reglas que le digan cómo hacerlo)
- las acciones indican los comandos a realizar para obtener el objetivo. Serán los pasos de compilación vistos en las secciones previas

NOTA: es importante tener en cuenta que el texto del archivo debe estar convenientemente sangrado usando tabuladores (no basta con espacios).

Iremos considerando una a una las reglas necesarias para generar el código anterior. Al aplicar una regla la herramienta **make** comprueba la lista de dependencias. Si están disponibles todos los archivos dependientes se ejecuta; en caso contrario se desencadena la ejecución de reglas que permitan generar las dependencias que estuviesen ausentes. También se analizan las fechas de los archivos a la hora de considerar válida una dependencia. Si un archivo con extensión **cpp** tiene fecha más reciente que su archivo objeto, entonces se considera que este último no está actualizado y será necesario volver a generarlo.

El contenido del archivo **Makefile** para el problema considerado es el siguiente:

```
1  obj/punto.o:  src/punto.cpp include/punto.h
2    g++ -c -o obj/punto.o -Iinclude src/punto.cpp
3
4  obj/circulo.o:  src/circulo.cpp include/circulo.h include/punto.h
5    g++ -c -o obj/circulo.o -Iinclude src/circulo.cpp
6
7  obj/utilidades.o:src/utilidades.cpp include/utilidades.h include/
       circulo.h include/punto.h
8    g++ -c -o obj/utilidades.o -Iinclude src/utilidades.cpp
9
10 lib/libformas.a: obj/punto.o obj/circulo.o obj/utilidades.o
11   ar rsv lib/libformas.a obj/punto.o obj/circulo.o obj/utilidades.o
12
13 obj/principal.o:  src/principal.cpp include/punto.h include/circulo.h
       include/utilidades.h
14   g++ -c -o obj/principal.o -Iinclude src/principal.cpp
15
16 bin/principal:  obj/principal.o lib/libformas.a
17   g++ -Llib/ -o bin/principal obj/principal.o -lformas
```

Comentamos una a una las reglas:

- regla de generación de **punto.o**: indica que deben procesarse los archivos que figuran en la lista de dependencias (**punto.cpp** y **punto.h**). Esta dependencia refleja que el archivo **punto.cpp** incluye (mediante la directiva **include**) el contenido de

punto.h. De esta forma, la regla se disparará en caso de que haya alguna modificación en cualquiera de ellos. La segunda línea contiene la acción necesaria para compilar el código fuente y producir el archivo objetivo, **punto.o**. Observar que la definición de objetivos y dependencias incluyen la referencia a los directorios donde se ubicarán los archivos de la aplicación

- regla de generación de **circulo.o**: la lista de dependencias incluye a **circulo.cpp**, **circulo.h** y **punto.h** (si se examina el código de **circulo.h** se observa que se incluye el archivo de cabecera de **punto.h**; de esta forma estos dos archivos son precisos para la compilación de este objetivo)

- regla de generación de **utilidades.o**: permite la compilación del archivo de utilidades, teniendo en cuenta las dependencias de las clases **Punto** y **Circulo**, así como del correspondiente archivo de código fuente (**utilidades.cpp**)

- regla de generación de la biblioteca **libformas.a**: a partir de los archivos objeto de las clases y del archivo de utilidades. La acción asociada a la regla es la orden usada previamente para generar la biblioteca

- generación de **principal.o**: con dependencias a los archivos de cabecera de las clases y de las funciones de utilidad (y también al propio archivo con el programa principal)

- regla de generación del ejecutable final: depende del archivo de código objeto **principal.o** y de la librería. La acción desencadenada por esta regla debe indicar el directorio de ubicación de las librerías (con la opción **-L**) así como la dependencia de la librería generada (mediante **-lformas**). Se observa que la indicación de la librería prescinde del prefijo **lib** y sólo precisa indicar el sufijo **formas**

Ejecución de make

Si ejecutamos el comando **make** en el directorio en que hemos ubicado el archivo **Makefile** (es decir, en el directorio base de la aplicación) se produce la siguiente salida:

```
mgomez@poldo:paso5> make
g++ -c -o obj/punto.o -Iinclude src/punto.cpp
```

Se aprecia que sólo se ejecuta la primera regla y nada más. Esto es así ya que la generación de **punto.o** se resuelve de forma directa a partir de los archivos de dependencia indicados en la regla y no es necesario disparar ninguna otra regla. ¿Cómo podemos conseguir que se ejecute la última regla, la que nos interesa para generar la aplicación? Para ello podemos pasar a **make** un argumento adicional que le indique la regla de interés:

```
mgomez@poldo:paso5> make bin/principal
g++ -c -o obj/principal.o -Iinclude src/principal.cpp
g++ -c -o obj/circulo.o -Iinclude src/circulo.cpp
g++ -c -o obj/utilidades.o -Iinclude src/utilidades.cpp
ar rsv lib/libformas.a obj/punto.o obj/circulo.o obj/utilidades.o
r - obj/punto.o
r - obj/circulo.o
r - obj/utilidades.o
g++ -Llib/ -o bin/principal obj/principal.o -lformas
```

Viendo la salida se comprueba que se han usado todas las reglas del archivo **Makefile** excepto la necesaria para generar **punto.o**, ya que este objetivo había sido generado por la llamada anterior (donde no se había especificado ningún argumento en la llamada a **make**).

Hay situaciones en que podemos desear usar un mismo archivo **Makefile** para generar dos aplicaciones diferentes. En este caso, la única solución posible consistirá en hacer dos llamadas a **make**, pasando en cada una de ellas el argumento correspondiente. Esto puede resolverse definiendo un objetivo especial, justo al principio del archivo, que suele denominarse **all** y que sólo incluye dependencias: aquellas asociadas a los objetivos que quieren generarse. En nuestro caso bastaría con agregar al archivo **Makefile** la siguiente línea, al principio:

```
all:  bin/principal
```

Una vez hecho esto la generación de la aplicación no precisa indicar argumento alguno:

```
mgomez@poldo:paso5> make
make: No se hace nada para 'all'.
```

Y como no ha habido cambio alguno, la herramienta determina que no es preciso ejecutar ninguna de las reglas. Imaginemos ahora que introducimos un pequeño cambio en **punto.h** (basta en realidad con que lo guardemos de nuevo, pese a no haberlo modificado en absoluto). Esto hará que la herramienta **make** detecte que el archivo modificado es más reciente que el archivo (o archivos) que se generan a partir de él. Esto obliga a que se ejecuten de nuevo las reglas necesarias para que la aplicación se construya de forma completa:

```
mgomez@poldo:paso5> make
g++ -c -o obj/principal.o -Iinclude src/principal.cpp
g++ -c -o obj/punto.o -Iinclude src/punto.cpp
g++ -c -o obj/circulo.o -Iinclude src/circulo.cpp
g++ -c -o obj/utilidades.o -Iinclude src/utilidades.cpp
ar rsv lib/libformas.a obj/punto.o obj/circulo.o obj/utilidades.o
r - obj/punto.o
r - obj/circulo.o
r - obj/utilidades.o
g++ -Llib/ -o bin/principal obj/principal.o -lformas
```

De la misma manera podemos ver qué ocurre si la modificación se produce en **circulo.h**:

```
mgomez@poldo:paso5> make
g++ -c -o obj/principal.o -Iinclude src/principal.cpp
g++ -c -o obj/circulo.o -Iinclude src/circulo.cpp
g++ -c -o obj/utilidades.o -Iinclude src/utilidades.cpp
ar rsv lib/libformas.a obj/punto.o obj/circulo.o obj/utilidades.o
r - obj/punto.o
r - obj/circulo.o
r - obj/utilidades.o
g++ -Llib/ -o bin/principal obj/principal.o -lformas
```

En este caso se comprueba que no se ha activado la regla necesaria para volver a generar **punto.o**. Igual ocurre si la modificación se produce en un archivo de código fuente. Por ejemplo, imaginemos que se produce un cambio en **utilidades.cpp**. Al volver a ejecutar el comando **make** se obtiene la siguiente salida:

```
mgomez@poldo:paso5> make
g++ -c -o obj/utilidades.o -Iinclude src/utilidades.cpp
ar rsv lib/libformas.a obj/punto.o obj/circulo.o obj/utilidades.o
r - obj/punto.o
r - obj/circulo.o
r - obj/utilidades.o
g++ -Llib/ -o bin/principal obj/principal.o -lformas
```

Este cambio ha precisado volver a generar **utilidades.o**, lo que implica a su vez la necesidad de volver a generar la biblioteca y la aplicación completa.

También suele ser habitual necesitar limpiar los archivos temporales (objetos y librerías) cuando ya no sean precisos. Suelen considerarse dos niveles de limpieza: sólo de archivos temporales y completa. Por esta razón se suelen agregar a los archivos de órdenes dos reglas adicionales, tras el objetivo **all**:

```
clean:
  echo "Limpieza archivos objeto y biblioteca...."
  rm obj/*.o lib/*.a

mrproper: clean
  rm bin/principal
```

La segunda de las órdenes limpiaría tanto los archivos de código objeto como la aplicación en sí. Debe observarse que la segunda regla incorpora como dependencia la ejecución de la primera. Estas dos órdenes se ejecutan cuando se pasan como argumento en la llamada a **make**:

```
make clean
make mrproper
```

Uso de macros

En el archivo **Makefile** ha sido necesario especificar los directorios de ubicación de archivos de cabecera, objeto y binario. Es posible mejorar un poco el contenido del archivo definiendo macros que indiquen los directorios de ubicación de los recursos: archivos de cabecera, ficheros objeto, librerías y ejecutable final. Esto permitiría cambiar la organización del código (por ejemplo, llamando a los directorios del proyecto de otra forma) y conseguir que todo funcione de forma correcta sin más que realizar unos cambios simples en el archivo **Makefile**.

```
1
2  OBJ=obj
3  BIN=bin
4  LIB=lib
5  SRC=src
6  INC=include
7
8  all:  bin/principal
9
10 $(OBJ)/punto.o: $(SRC)/punto.cpp $(INC)/punto.h
11   g++ -c -o $(OBJ)/punto.o -I$(INC) $(SRC)/punto.cpp
12
13 $(OBJ)/circulo.o: $(SRC)/circulo.cpp $(INC)/circulo.h $(INC)/punto.h
14   g++ -c -o $(OBJ)/circulo.o -I$(INC) $(SRC)/circulo.cpp
15
16 $(OBJ)/utilidades.o:  $(SRC)/utilidades.cpp $(INC)/utilidades.h $(INC)/punto.h $(
     INC)/circulo.h
```

```
17    g++ -c -o $(OBJ)/utilidades.o -I$(INC) $(SRC)/utilidades.cpp
18
19  $(LIB)/$(LIB)formas.a: $(OBJ)/punto.o $(OBJ)/circulo.o $(OBJ)/utilidades.o
20    ar rsv $(LIB)/$(LIB)formas.a $(OBJ)/punto.o $(OBJ)/circulo.o $(OBJ)/utilidades.o
21
22  $(OBJ)/principal.o: $(SRC)/principal.cpp $(INC)/punto.h $(INC)/circulo.h $(INC)/
        utilidades.h
23    g++ -c -o $(OBJ)/principal.o -I$(INC) $(SRC)/principal.cpp
24
25  $(BIN)/principal: $(OBJ)/principal.o $(LIB)/libformas.a
26    g++ -L$(LIB)/ -o bin/principal $(OBJ)/principal.o -lformas
```

En este ejemplo se han proporcionado macros para los 5 directorios usados. La declaración de las macros se hace al inicio del archivo y se usan incluyendo el nombre de la macro entre paréntesis y precedido por el símbolo **$**.