Guion de prácticas

Compilación separada (2)

Febrero de 2018

UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2017/2018

Índice

1. Introducción al guion

Este guión pretende continuar con la metodología de la compilación separada de problemas, en particular, con el uso de la herramienta **make**, por lo que se continuará a partir de la Versión 4 del guión anterior. Para más detalles sobre el proceso de compilación y ejecución en Linux y el uso de `make`, consulte el Guión "Introducción a la Compilación de Programas en C++".

2. Versión 5. Automatización de la compilación con ficheros Makefile

Cree una nueva carpeta llamada `Version5` donde haremos una nueva versión del programa a partir de la versión 4 (misma estructuración en carpetas) copiando los ficheros en las mismas carpetas.

En esta versión construiremos un fichero **makefile** para gestionar automáticamente la compilación de la aplicación. El archivo **makefile** se construirá con un editor de texto, y se colocará en la carpeta padre de `src` (ver Figura 1).

```
./
├── makefile
├── bin
├── data
│   └── circulomedio.dat
├── doc
│   └── circulomedio.doxy
├── include
│   ├── circulo.h
│   └── punto.h
├── lib
├── obj
└── src
    ├── central.cpp
    ├── circulo.cpp
    └── punto.cpp
```

Figura 1: Estructura básica de las carpetas de un proyecto con múltiples módulos antes de compilar y enlazar

Para facilitar el desarrollo de la práctica, así como el estudio de las dependencias que se reflejan en el archivo **makefile**, se proporciona en la Figura 1 un esquema de los archivos y sus relaciones.

Puesto que conocemos las dependencias entre los ficheros de nuestro proyecto, tras hacer modificaciones en un módulo no necesitamos reconstruir el proyecto completo. Cuando modificamos algún fichero sólo es necesario reconstruir los ficheros que dependen de él, lo que hace que a su vez sea necesario reconstruir los ficheros que dependan de los nuevos

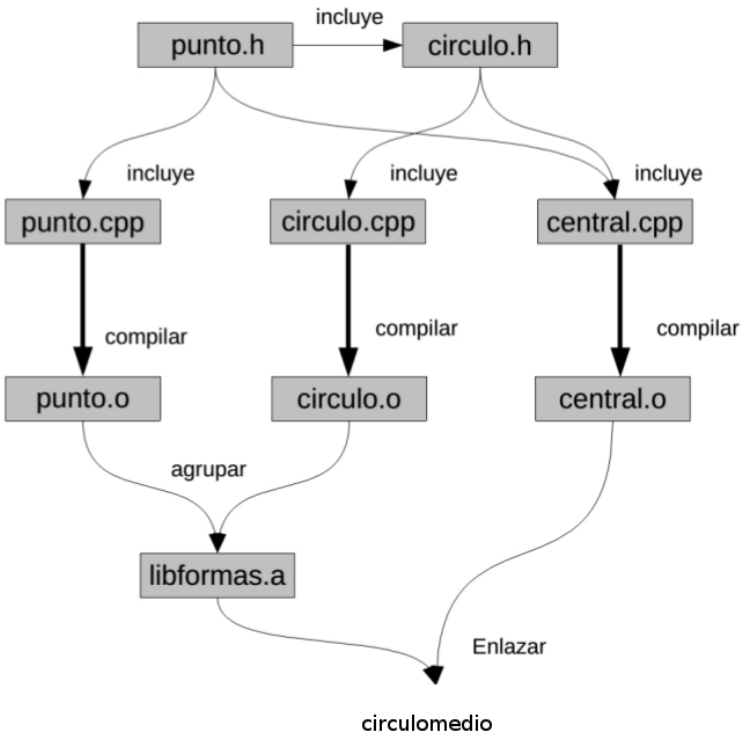

Figura 2: Módulos y relaciones para el programa “circulomedio”. Las flechas indican dependencias: (1) Las dependencias de inclusión se deben añadir tanto si es inclusión directa como indirecta y (2) El resto de dependencias se especifican sólo si son directas.

ficheros reconstruidos, y así sucesivamente. El problema es que, dependiendo del módulo al que afecten las modificaciones, lo que tenemos que reconstruir podrá variar. Por ejemplo, si modificamos circulo.cpp debemos reconstruir `circulo.o`, `libformas.a`, y el ejecutable `circulomedio`. Sin embargo, si modificamos `central.cpp` sólo tendremos que reconstruir `central.o` y el ejecutable.

Para evitar acordarnos de qué partes se han de reconstruir en función de cuales se han modificado, disponemos de herramientas que automatizan todo el proceso. En nuestro caso usaremos la utilidad **make**. Su cometido es leer e interpretar un fichero en el que se almacenan una serie de reglas que reflejan el esquema anterior de dependencias. Lo habitual es que a este fichero de reglas se le llame `Makefile` o `makefile`, aunque podría cambiar. El aspecto de una regla es el siguiente:

```
1 Objetivo : Lista de dependencias
2     Acciones
```

Donde:

- `Objetivo`: Esto es lo que queremos construir. Habitualmente es el nombre de un fichero que se obtendrá como resultado del procesa-

miento de otros ficheros. Más adelante veremos que no tiene que ser, necesariamente, un nombre de fichero.

- `Lista de dependencias`: Esto es una lista de items de los que depende la construcción del objetivo de la regla. Normalmente los items son ficheros o nombres de otros objetivos. Al dar esta lista de dependencias, la utilidad make debe asegurarse de que han sido todas satisfechas antes de poder alcanzar el objetivo de la regla.
- `Acciones`: Este es el conjunto de acciones que se deben llevar a cabo para conseguir el objetivo. Normalmente serán instrucciones como las que hemos visto antes para compilar, enlazar, etc.

IMPORTANTE: Observe que al escribir estas reglas en el fichero Makefile, las acciones se deben sangrar (o tabular) con un carácter tabulador. Si se rellena con espacios, el programa **make** no es capaz de entender la regla.

Por ejemplo, la siguiente sería una regla válida:

```
1 bin/circulomedio : obj/central.o lib/libformas.a
2     g++ -o bin/circulomedio obj/central.o -Llib/ -lformas
```

donde el objetivo es la construcción del ejecutable `bin/circulomedio`. En la lista de dependencias hemos puesto la lista de ficheros de los que depende, o lo que es lo mismo, estamos diciendo que para poder construir `bin/circulomedio`, previamente han de haber sido construidos adecuadamente los ficheros `obj/central.o` y `lib/libformas.a`. Finalmente la acción que hay que llevar a cabo para generar el objetivo es la ejecución de **g++** con los parámetros que vemos en el ejemplo.

Si al aplicar la regla, se detecta que alguno de los items de la lista de dependencias no existe o necesita ser actualizado, entonces se ejecutarán las reglas necesarias para crearlo de nuevo antes de aplicar las acciones de la regla actual.

En realidad, **make** toma uno por uno los items de la lista de dependencias e intenta buscar alguna regla que le diga cómo se debe construir ese item. De esta forma, si por ejemplo está analizando el item `lib/libformas.a`, se buscará una regla cuyo objetivo sea ese. En nuestro ejemplo, esa regla debería ser la siguiente:

```
1 lib/libformas.a: obj/punto.o obj/circulo.o
2     ar rsv lib/libformas.a obj/punto.o obj/circulo.o
```

de esta forma, para saber si `lib/libformas.a` necesita ser actualizado se aplicará la regla de forma análoga a como hicimos con la regla anterior. Al analizar la lista de dependencias, intentará comprobar si los items `obj/punto.o` y `obj/circulo.o` necesitan ser actualizados y buscará reglas con esos objetivos. En nuestro ejemplo las reglas podrían ser éstas:

```
1 obj/punto.o: src/punto.cpp include/punto.h
2     g++ -c src/punto.cpp -o obj/punto.o -Iinclude/
3
4 obj/circulo.o: src/circulo.cpp  include/circulo.h include/punto.h
5     g++ -c src/circulo.cpp -o obj/circulo.o -Iinclude/
```

Ahora, la lista de dependencias para la regla `obj/punto.o` contiene los items `src/punto.cpp` y `include/punto.h`. Al tratarse de ficheros de código fuente, no será frecuente que haya reglas cuyo objetivo sea obtener éstos, ya que esos ficheros no se van a crear a partir de otros ficheros sino que serán creados manualmente en un editor de textos por parte del programador. En esta situación, para averiguar si es necesario volver a construir el objetivo `obj/punto.o`, se comprobará si se da o no alguna de las siguientes situaciones:

- Que el objetivo no exista. Esta situación indica que el código fuente no fué compilado con anterioridad y que, por lo tanto, se ejecutarán las acciones de la regla para compilarlo y generar el código objeto.
- Que el objetivo ya exista pero su fecha de modificación sea anterior a la del fichero de código fuente. En este caso es evidente que, aunque con anterioridad ya fué compilado el código fuente y fué creado el código objeto, en algún momento posterior el programador ha modificado dicho código fuente. Esto implica que el código objeto necesita ser actualizado, es decir, que hay que aplicar las acciones de la regla para volver a compilarlo.
- Si no se da ninguna de las dos situaciones anteriores, entonces es que el objetivo está actualizado y no se aplicarán las acciones.

Misma forma de proceder para el objetivo `obj/central.o`.

```
1 obj/central.o: src/punto.cpp include/punto.h  include/circulo.h
2    g++ -c src/central.cpp -o obj/central.o -Iinclude
```

Por tanto, al aplicar la primera regla (`bin/circulomedio`), si por ejemplo hemos hecho cambios en `src/punto.cpp`, se aplicarán de manera sucesiva (y ordenada) todas las reglas necesarias para ir construyendo todos los ficheros intermedios del proceso de compilación y enlazado. El encadenamiento de reglas lo gestiona **make** de manera automática y no debemos preocuparnos por el orden en el que escribimos las reglas en el fichero `Makefile`; él sabe buscar la regla adecuada en cada momento.

2.1. Ejecutando make

Si ejecutamos **make** sin ningún parámetro, intentará aplicar la primera regla que se encuentre en el fichero `Makefile`. Si queremos aplicar otra regla, debemos llamar a **make** usando como parámetro el objetivo de dicha regla. Por ejemplo, en el caso anterior, y suponiendo que la regla `bin/circulomedio` no es la primera que hemos escrito (recuerda que el orden de las reglas en `Makefile` no es importante), debemos ejecutar:

```
make bin/circulomedio
```

Esta ejecución de **make** construirá la aplicación **circulomedio** y todos los ficheros necesarios para conseguir dicho objetivo.

A veces queremos que el fichero `Makefile` construya más de una aplicación. Por ejemplo, suponer que además de querer construir la aplicación **circulomedio** quisiéramos contruir también la aplicación **main2**. En ese caso, una vez añadida al `Makefile` la regla para construir esta segunda aplicación, podríamos llamar a **make** dos veces para construir las dos aplicaciones:

```
make bin/circulomedio
make bin/main2
```

Si esto es lo que tenemos que hacer siempre para crear nuestro proyecto, es habitual incluir en el makefile una nueva regla que obligue a la construcción de nuestras dos aplicaciones. Esta regla, que sirve para agrupar todos los objetivos del proyecto, se suele escribir al comienzo de `Makefile` (la primera regla) y se llama `all` (ese nombre es el objetivo de la regla). La forma concreta sería:

```
1 all : bin/circulomedio bin/main2
```

Observe que no tiene acciones asociadas. Al haber sido escrita en primer lugar en `Makefile`, bastará con ejecutar:

```
make
```

para que sea aplicada. Al aplicarla, lo primero que se intenta hacer es verificar si es necesario actualizar alguna de sus dependencias, que son los ejecutables **circulomedio** y **main2**. Si aún no han sido generados, se crean (aplicando otras reglas). Una vez que ha terminado ese proceso, habría que pasar a aplicar las acciones de esta regla, pero como no tiene, no se hace nada más. El resultado es que ejecutando simplemente **make** hemos conseguido crear todos los ejecutables del proyecto.

2.2. Otras reglas estándar

A veces se hace necesario borrar ficheros que se consideran temporales o ficheros que no se van a necesitar con posterioridad. Por ejemplo, una vez acabado el proyecto definitivamente, no serán necesarios los códigos objeto ni, probablemente, las bibliotecas por lo que podemos borrarlos.

En este sentido se suelen incorporar algunas reglas que hacen estas tareas de limpieza. Es frecuente considerar dos niveles de limpiado del proyecto. Un primer nivel que limpia ficheros intermedios pero deja las aplicaciones finales que hayan sido generadas, `clean_blando`. Y un segundo nivel `clean` que elimina todos los archivos binarios, lo que permite migrar una aplicación a otra máquina.

```
1 clean:
2     @echo "Limpiando..."
3     rm obj/*.o lib/*.a bin/*
```

De esta forma, tras acabar de programar el proyecto podemos quedarnos únicamente con el código fuente original y los ejecutables. Observe que esta regla no tiene dependencias por lo que al aplicarla, directamente se pasa a ejecutar sus acciones.

Otras reglas estándar podrían dedicarse a la generación automática de la documentación o al proceso de limpiar y comprimir el proyecto, listo para la entrega.

```
1 doxy:
2         doxygen doc/circulomedio.doxy
3
4 zip: clean
5         zip -r zip/practica2.zip *
```

2.3. Consideraciones sobre los ficheros de cabecera

Los ficheros de cabecera también deben ser tenidos en cuenta a la hora de escribir las reglas para **make**. Por ejemplo, en el caso de la regla que dice como se construye el código objeto del módulo (`obj/punto.o`), debemos poner en la lista de dependencias todos aquellos ficheros que afectan a la construcción de ese objetivo de tal forma que, si alguno de ellos fuese modificado, se obligase a la reconstrucción de dicho objetivo. En particular, si vemos el contenido de `src/punto.cpp`, podemos apreciar que este hace un `#include` de `punto.h`, por lo que la regla debe tenerlo en cuenta (esta dependencia también queda clara en la Figura 2):

```
1 obj/punto.o: src/punto.cpp include/punto.h
2     g++ -c src/punto.cpp -o obj/punto.o -Iinclude/
```

2.4. Obtención automática de las dependencias

El preprocesador de **g++** es capaz de determinar todas las dependencias necesarias para la compilación de un determinado módulo. Además, como resultado, proporciona la cabecera de la regla necesaria para el correspondiente fichero `Makefile`. Para esto se usa la opción `-MM` en la llamada a **g++**. Siguiendo con el ejemplo de esta práctica, si ejecutamos:

```
g++ -MM -Iinclude src/central.cpp
```

obtenemos como resultado:

```
central.o: src/central.cpp include/punto.h \
     include/circulo.h
```

Observe que se incluyen como dependencias todos los ficheros de cabecera que, directa o indirectamente, están siendo utilizados por `central.cpp`.

2.5. Versión 6. Uso de macros en makefiles

Uno de los problemas que podemos ver en el fichero **makefile** anterior, 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? Tendríamos que cambiar una por una todas las ocurrencias de dichos directorios. Mediante el uso de *macros*, que son variables o cadenas que se expanden cuando realizamos una llamada a **make**, conseguimos que los nombres de los directorios sólo los tengamos que escribir una vez. Por ejemplo, incluiremos al principio de nuestro **makefile**, una macro para el directorio `bin` de la forma `BIN = bin` y luego siempre que queramos hacer referencia al directorio `bin` escribiremos `$(BIN)`.

Modifique el anterior fichero makefile para que use macros de este tipo, en lugar de directamente los nombres de los directorios (Sección 5.2).

3. Práctica a entregar: Intervalos 2

Se pide continuar a partir de la Versión 4 del proyecto de intervalos de la práctica anterior y crear la versión 6 que incluya el makefile con macros tal y como se ha descrito en las secciones anteriores y se ilustra como ejemplo en la Sección 5. También se debe incluir una nueva función externa llamada `interseccion` que haga lo propio con los dos intervalos que recibe como parámetros. Esta función debe ser implementada como una función **friend** de la clase **Intervalo**.

```
class Intervalo{
private:
    ...
    bool validar(double , double , bool , bool );
public:
    ...
/**
 * @brief realiza la interseccion de dos intervalos, puede
 * resultar un intervalo vacío en caso de que no tengan cotas
 * comunes, en caso contrario se revisan las cotas.
 * @param i1 primer intervalo de entrada
 * @param i2 segundo intervalo de entrada
 * @return devuelve el intervalo resultante de realizar la
 * interseccion entre los dos intervalos de entrada
 */
friend Intervalo interseccion(const Intervalo &, const Intervalo &);
};
```

Las funciones externas definidas como **friend** de una clase tienen el privilegio de poder acceder a los datos miembro y métodos **private:** y **protected:** de una clase. En este caso, la ventaja de declarar esta función como **friend** es acceder al método **validar** para comprobar que el intervalo resultado que devuelve la función siempre es un intervalo correcto.

La intersección $\cap$ de dos intervalos resulta un intervalo. $i1 \cap i2 = ir$, dónde ir es un intervalo simple de los definidos en la clase intervalo, incluido el vacío. Para definir el intervalo resultado se comparan las cotas inferiores de $i1$ e $i2$, y se obtiene como cota inferior la mayor de las dos. La cota superior del resultado será la menor de las cotas superiores de

$i1$ e $i2$. Por último, en resultado se establece la propiedad de pertenencia de cada una de las cotas asignadas anteriormente en el resultado. Finalmente, el resultado puede ser un intervalo imposible (las cotas se cruzan, ...), en ese caso se devuelve un intervalo vacío[1].

Ejemplo:
$[0, 10] \cap (0, 10) = (0.,10)$
$[0, 10] \cap [1, 1] = [1, 1]$
$[9, 13] \cap (0, 10) = [9, 10)$
$[0, 5) \cap (5, 10] = (0, 0)$

Se pide también ampliar la función **main** para que calcule y muestre la intersección de cada dos intervalos consecutivos leídos desde el teclado, tal y como muestra el siguiente ejemplo.

```
$$ bin/intervalo < data/intervalo.dat

Cuantos intervalos se van a introducir? (max 10):
Introduce [ o ( cotaInferior, cotaSuperior ) o ]
Comprobando intervalo vacio
(0,0) Es vacío
[1,1] No es vacío

[0,10]
Valores dentro del intervalo:0 5.7 9.6 10
Valores fuera del intervalo:-1 -0.001
(0,10]
Valores dentro del intervalo:5.7 9.6 10
Valores fuera del intervalo:-1 -0.001 0
[0,10)
Valores dentro del intervalo:0 5.7 9.6
Valores fuera del intervalo:-1 -0.001 10
(0,10)
Valores dentro del intervalo:5.7 9.6
Valores fuera del intervalo:-1 -0.001 0 10
La intersección de [0,10] y de (0,10] es (0,10]
La intersección de (0,10] y de [0,10) es (0,10)
La intersección de [0,10) y de (0,10) es (0,10)
```

Para ello, se proporciona en DECSAI un nuevo fichero de datos de validación **intervalo2.dat** cuya salida asociada es la siguiente.

```
$$ bin/intervalo < data/intervalo2.dat

Cuantos intervalos se van a introducir? (max 10):
Introduce [ o ( cotaInferior, cotaSuperior ) o ]
Comprobando intervalo vacio
(0,0) Es vacío
[1,1] No es vacío

[0,5]
Valores dentro del intervalo:0
Valores fuera del intervalo:-1 -0.001 5.7 9.6 10
(5,10]
```

[1] Caso en que no tengan ningun punto común, el intervalo es el vacío.

```
Valores dentro del intervalo:5.7 9.6 10
Valores fuera del intervalo:-1 -0.001 0
La intersección de [0,5] y de (5,10] es (0,0)
```

4. Entrega

Se debe de entregar através de **DECSAI** un fichero zip, **practica2.zip** la estructura de directorios ya expuesta: **bin, data, include, lib, obj, src, doc** y **comprimida con la regla zip que aparece en el makefile de la sección 2.2**.

```
./
├── makefile
├── include
│   └── intervalo.h
├── src
│   ├── intervalo.cpp
│   └── main.cpp
├── bin
├── obj
├── lib
├── data
│   ├── intervalo.dat
│   └── intervalo2.dat
├── doc
│   └── intervalo.doxy
└── zip
```

4.1. Corrección de la práctica

La práctica se corregirá con la siguiente secuencia de comandos.

```
$$  unzip practica2.zip
Archive:  practica2.zip
   creating: bin/
   creating: data/
  inflating: data/intervalo.dat
 extracting: data/intervalo2.dat
   creating: doc/
  inflating: doc/intervalo.doxy
   creating: include/
  inflating: include/intervalo.h
   creating: lib/
  inflating: makefile
   creating: obj/
   creating: src/
  inflating: src/main.cpp
  inflating: src/intervalo.cpp
   creating: zip/

$$ make
g++ -Wall -g -c ./src/main.cpp -o ./obj/main.o -I./include
g++ -Wall -g -c ./src/intervalo.cpp -o ./obj/intervalo.o -I...
ar -rvs ./lib/libintervalo.a ./obj/intervalo.o
ar: creando ./lib/libintervalo.a
a - ./obj/intervalo.o
g++ -o ./bin/intervalo ./obj/main.o -lintervalo -L./lib

$$ bin/intervalo < data/intervalo2.dat

Cuantos intervalos se van a introducir? (max 10):
Introduce [ o ( cotaInferior, cotaSuperior ) o ]
Comprobando intervalo vacio
(0,0) Es vacío
[1,1] No es vacío

[0,5]
Valores dentro del intervalo:0
Valores fuera del intervalo:-1 -0.001 5.7 9.6 10
(5,10]
Valores dentro del intervalo:5.7 9.6 10
Valores fuera del intervalo:-1 -0.001 0
La intersección de [0,5] y de (5,10] es (0,0)
```

5. Ejemplos de makefiles para circulomedio

5.1. Sin macros

```
bin/circulomedio : obj/central.o lib/libformas.a
        g++ -o bin/c obj/central.o -lformas -Llib

obj/central.o : src/central.cpp include/punto.h include/circulo.h
        g++ -c src/central.cpp -o obj/central.o -Iinclude

obj/punto.o : src/punto.cpp include/punto.h
        g++ -c src/punto.cpp -o obj/punto.o -Iinclude

obj/circulo.o : src/circulo.cpp include/circulo.h include/punto.h
        g++ -c src/circulo.cpp -o obj/circulo.o -Iinclude

lib/libformas.a : obj/circulo.o obj/punto.o
        ar -rvs lib/libformas.a obj/punto.o obj/circulo.o

clean:
        rm -f obj/* bin/* lib/*
```

5.2. Con macros

```
# Definición de macros para definir las carpetas de trabajo
BIN=./bin
OBJ=./obj
SRC=./src
INC=./include
LIB=./lib
ZIP=./zip
DOC=./doc
# Opciones de compilación
# -Wall muestra todas las advertencias
# -g compila en modo "depuración"
OPT=-Wall -g
# Nombre de la práctica
PRJ=practica2

# Las macros se usan en las reglas del makefile como si fuesen variables
# que se sustituyen por su valor definido anteriormente
all : $(BIN)/circulomedio
$(BIN)/circulomedio : $(OBJ)/central.o lib/libformas.a
        g++ -o $(BIN)/circulomedio $(OBJ)/central.o -lformas -L$(LIB)

$(OBJ)/punto.o : $(SRC)/punto.cpp $(INC)/punto.h
        g++ $(OPT) -c $(SRC)/punto.cpp -o $(OBJ)/punto.o -I$(INC)

$(OBJ)/circulo.o : $(SRC)/circulo.cpp $(INC)/circulo.h $(INC)/punto.h
        g++ $(OPT) -c $(SRC)/circulo.cpp -o $(OBJ)/circulo.o -I$(INC)

$(OBJ)/central.o : $(SRC)/central.cpp $(INC)/punto.h $(INC)/circulo.h
        g++ $(OPT) -c $(SRC)/central.cpp -o $(OBJ)/central.o -I$(INC)

$(LIB)/libformas.a : $(OBJ)/punto.o $(OBJ)/circulo.o
        ar -rvs $(LIB)/libformas.a $(OBJ)/punto.o $(OBJ)/circulo.o
```

```
clean:
        @echo "Limpiando ... "
        rm -rf $(OBJ)/*.o $(BIN)/* $(LIB)/* $(ZIP)/* $(DOC)/latex $(DOC)/html

zip: clean
        @echo "Generando ZIP del proyecto " $(PRJ)
        zip -r $(ZIP)/$(PRJ).zip *

doxy:
        doxygen doc/circulomedio.doxy
```