Guion de prácticas 1

Modularización y compilación separada

DECSAI

Metodología de la Programación

Grado en Ingeniería Informática

Prof. David A. Pelta

Índice

1. Introducción al guion

Para esta sesión de prácticas, el alumno deberá entender los conceptos relacionados con la compilación separada (ver tema 11 del libro Garrido, A. "*Fundamentos de programación con la STL*", Editorial Univ. de Granada, 2016).

Vamos a utilizar dos clases. Una clase *Punto2D* para representar puntos en el plano, y una clase *Circulo* que contendrá el centro del mismo (un punto en el plano) y el radio.

El objetivo final será la creación de dos módulos *punto* y *circulo* (es decir, 4 archivos considerando que cada uno tiene un fichero cabecera *.h* y uno de implementación *.cpp*). Estos módulos implementan la funcionalidad de dos clases. Por un lado, la clase *Punto2D*, que contendrá dos datos miembro de tipo *double* para las coordenadas *x* e *y*. Por otro lado, la clase *Circulo*, que contendrá como datos miembro un objeto de la clase *Punto2D* para representar el centro, y un valor de tipo *double* para representar el radio.

Con el fin de centrarnos en la compilación separada, se proporciona el código que implementa un programa que debe crear dos objetos de tipo *Circulo*, C_1, C_2, y crear un tercero C_3 que tenga como centro, el punto medio entre los centros de C_1 y C_2, y cuyo radio sea la mitad de la diferencia entre los radios respectivos. El programa mostrará los tres círculos así como sus areas.

El código de los módulos, así como el programa de prueba lo puede descargar desde PRADO: `http://prado.ugr.es`. El contenido del fichero comprimido es el siguiente:

1. Módulo *Punto2D*: implementado en *punto.h* y *punto.cpp*. Contiene el código para manejar la clase *Punto2D*.
2. Módulo *Circulo*: implementado en *circulo.h* y *circulo.cpp*. Contiene el código para manejar la clase *Circulo*. Hace uso del módulo *Punto2D*.
3. Módulo *Central*: implementado en *central.cpp*. Contiene el código que implementa el programa de cálculo del círculo central y su área. Hace uso de los módulos *Punto2D* y *Circulo*.

Una vez completado el programa, compílelo[1].

2. Compilación separada: Versión 1

Cree una nueva carpeta llamada `practica1/version1`. Dentro de ella, deberá crear los directorios *include* (para los ficheros `.h`), *src* (para los ficheros `.cpp`), *obj* (para los ficheros `.o`) y *bin* (para los ejecutables). A continuación, ubique los ficheros descargados en las carpetas que corresponda.

[1] Si ha utilizado la función *to_string()*, deberá añadir la opción -std=c++0x luego del nombre del fichero fuente.

Observe que para evitar dobles inclusiones, los ficheros `.h` contienen directivas del preprocesador de la forma siguiente:

```
#ifndef _FICHERO_H_
#define _FICHERO_H_
...
#endif
```

Los ficheros `.h` incluyen la definición de las clases y las cabeceras de los métodos. Por ejemplo, el contenido de `punto.h` es:

```
#ifndef _PUNTO_H_
#define _PUNTO_H_
class Punto2D {
private:
    double x;
    double y;

public:
   Punto2D();
   Punto2D(double px, double py);
   double getX();
   double getY();
  -------------------
}
#endif
```

En los ficheros `.cpp` se incluye la implementación de los métodos correspondientes. Los ficheros `.h` deberán incluirse donde sea necesario, con la sentencia:

```
#include "fichero.h"
```

Cuando tenga los tres módulos, se deberán compilar para obtener los archivos *punto.o*, *circulo.o* y *central.o*. Para ello, situese en el directorio padre de las carpetas *src*, *include*, etc.

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

Una vez que dispone de los tres archivos objeto, *punto.o*, *circulo.o* y *central.o*, deberá enlazarlos para obtener el ejecutable *central* de la aplicación propuesta.

```
g++ obj/central.o obj/punto.o obj/circulo.o -o bin/central
```

Ejecute el nuevo programa para comprobar el funcionamiento.

Si modificamos el fichero `punto.h`, ¿qué órdenes sería necesario volver a ejecutar para obtener de nuevo el ejecutable? ¿Y si modificamos `circulo.cpp`?

3. Creación de una biblioteca: Versión 2

Copie toda la carpeta `version1` a otra `version2`. Dentro de esta, cree una carpeta `lib`. En esta sesión deberá crear una biblioteca. Concretamente, debe realizar las siguientes tareas:

1. Compile los ficheros `.cpp`.

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

2. Cree una biblioteca con los archivos *punto.o* y *circulo.o*, con nombre *libformas.a*.

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

3. Ejecute la orden para crear el ejecutable *central* teniendo en cuenta esta biblioteca, es decir, sin enlazar directamente con los archivos objeto.

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

   Indique de nuevo qué debemos hacer si modificamos el fichero `punto.h`, ¿qué órdenes sería necesario volver a ejecutar para obtener de nuevo el ejecutable? ¿Y si modificamos `circulo.cpp`?

4. Automatización de la compilación con ficheros Makefile: Versión 3

Cree una nueva carpeta llamada `practica1/version3` donde haremos una nueva versión del programa a partir de la versión 2 (misma estructuración en carpetas) copiando los ficheros `.cpp` y `.h`.

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`. 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 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 `central`.

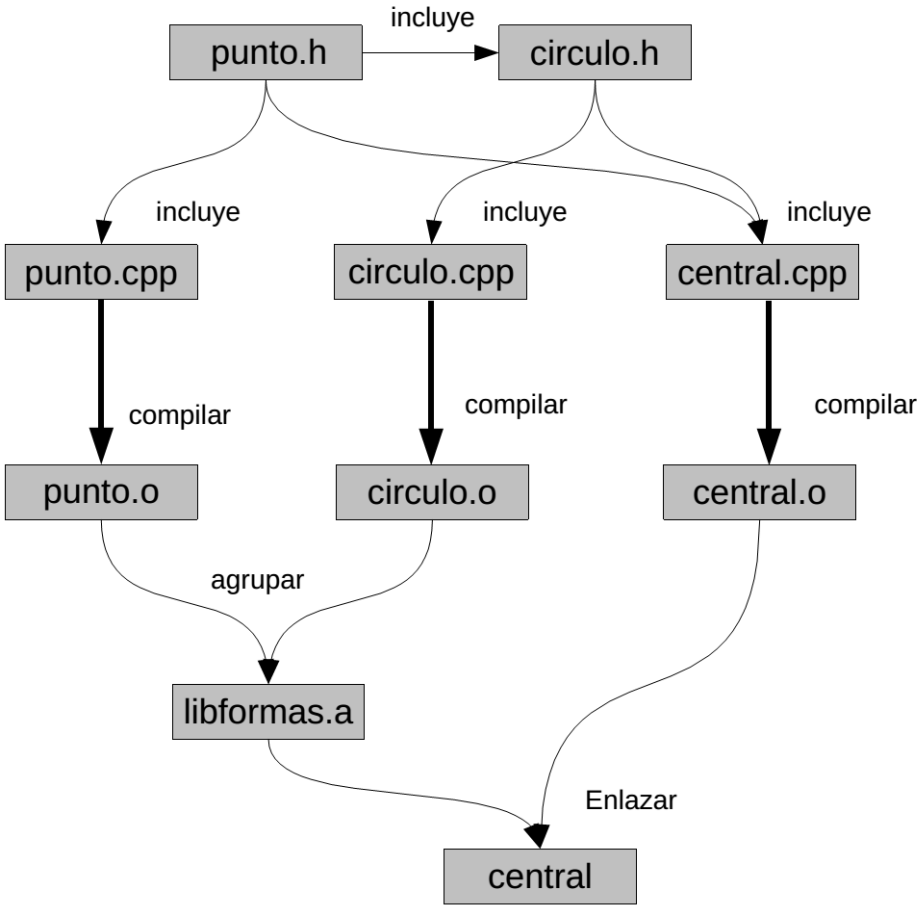

Figura 1: Módulos y relaciones para el programa “central”. Las flechas indican dependencias: (1) Las dependencias de inclusión se deben añadir tanto si es inclusión directa como indirecta y (2) El resto de dependencias se especifican sólo si son directas.

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 no, el programa **make** no es capaz de entender la regla.

Por ejemplo, la siguiente sería una regla válida:

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

donde el objetivo es la construcción del ejecutable `bin/central`. 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/central`, 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.

Por tanto, al aplicar la primera regla (`bin/central`), 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.

4.1. 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/central` no es la primera que hemos escrito (recuerda que el orden de las reglas en `Makefile` no es importante), debemos ejecutar:

```
make bin/central
```

Esta ejecución de **make** construirá la aplicación **central** 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 **central** 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/central
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/central 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 **central** 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.

4.2. 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:

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

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.

Hay un segundo nivel que, además de limpiar lo mismo que la regla anterior, también limpia los resultados finales del proyecto.

```
1 mrproper: clean
2     rm bin/central bin/main2
```

De esta forma lo que conseguimos es que se aplique la regla clean y que, a continuación, se apliquen las acciones de esta regla.

4.3. 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 1):

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

4.4. 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`.

4.5. 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.