Compilación separada y Gestión de proyectos con make

Antonio Garrido / Javier Martínez Baena

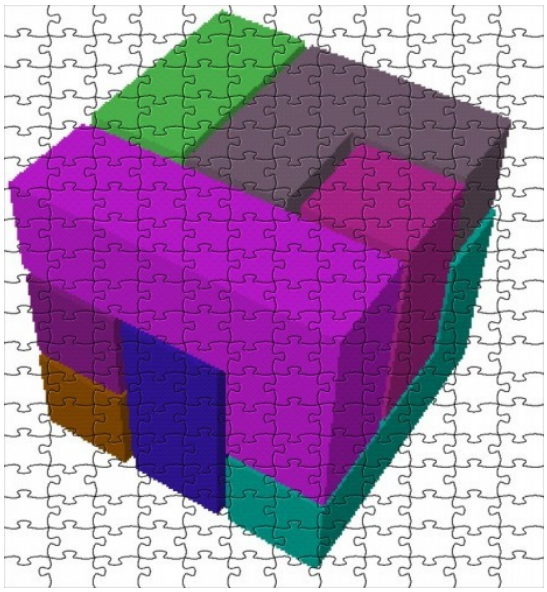
Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Metodología de la Programación

Grado en Ingeniería Informática

Índice de contenido

1.Introducción....3
1.1.Problema: círculo central....3
1.2.Compilación y ejecución....3
2.Módulos y compilación separada....4
2.1.Separación en archivos....5
2.2.Compilación y ejecución....5
2.3.Bibliotecas....6
2.3.1.Enlazar con bibliotecas....8
2.3.2.Orden de bibliotecas....8
3.Gestión del proyecto con make....9
3.1.Dependencias entre módulos....9
3.2.Archivo makefile....10
3.2.1.Reglas....11
3.2.2.Macros....12
3.2.3.Múltiples objetivos finales....13
3.2.4.Distribución en directorios....14
3.2.5.Otras reglas estándar....15
3.2.6.Reglas implícitas....15
4.Referencias....16



1. Introducción

Los objetivos de este guión de prácticas son los siguientes:

1. Crear un programa usando compilación separada con el compilador "*g++*" de la *GNU*.
2. Crear y usar bibliotecas. Conocer la orden "*ar*" y el uso de bibliotecas con "*g++*".
3. Aprender a manejar archivos "*makefile*" básicos. Entender cómo se puede gestionar automáticamente la compilación con la orden "*make*".

Los requisitos para poder realizar esta práctica son:

1. Saber dividir un programa en distintos módulos, creando archivos de cabecera ("*.h*") y compilables ("*.cpp*").

1.1. Problema: círculo central

Para desarrollar este guión práctico nos basaremos en un problema muy simple: el cálculo de un círculo central. El problema y la solución propuesta están diseñados para enfatizar los distintos aspectos que queremos dejar claros. Lógicamente, el lector podría haber propuesto otra solución para el mismo problema.

El problema consiste en leer dos círculos de entrada y calcular el círculo central que pasa por el centro de ambos (problema 12.1 de [GAR06a]). Para ello creamos un programa que incluye dos nuevos tipos de datos:

1. El tipo *Punto* que contiene dos objetos, de tipo *double*, para las coordenadas x e y correspondientes. Un objeto de este tipo representa un punto (x,y) en el espacio 2D.
2. El tipo *Circulo* que está compuesto de objetos, uno de tipo *Punto* para indicar el centro y otro de tipo *double*, para indicar el radio. Un objeto de este tipo representa un círculo radio-(x,y) en el espacio 2D.

Además, junto con estos tipos, se crean un conjunto de funciones que realizan operaciones con ellos. Así, podemos resolver:

1. E/S de un punto o un círculo desde/hacia la entrada/salida estándar.
2. Dados dos puntos, cuál es la distancia euclídea entre ellos.
3. Dados dos puntos, cuál es el punto medio entre ellos.
4. Dado un punto y un círculo, determinar si el punto está en el interior del círculo.
5. Dado un círculo, obtener el área del círculo.
6. Etc.

Con estas funciones, y los nuevos tipos, nos resulta bastante simple resolver el problema del círculo central.

1.2. Compilación y ejecución

Inicialmente, podemos tener el programa escrito completamente en un mismo archivo: por ejemplo, *central.cpp*. Por tanto, en éste estarán las definiciones de los tipos, las funciones que operan con ellos, y la función *main* que resuelve el problema del círculo central.

La compilación del programa se puede realizar de una forma muy sencilla en una única línea, indicando al compilador que obtenga el ejecutable correspondiente:

```
g++ -o central central.cpp
```

Observe que hemos llamado al compilador (g++) indicando dos cosas:

1. El nombre del archivo resultante, con la opción "-o".
2. El nombre del archivo a compilar, es decir, el código fuente "central.cpp".

En principio, el orden de los argumentos que se pasan al programa "*g++*" no es importante, y se puede cambiar. Ahora bien, es importante tener en cuenta que el nombre del resultado vendrá, obligatoriamente, después de la opción "-o". En caso de no hacerlo, por ejemplo, intercambiando el orden de "*central*" y "*central.cpp*", el compilador podría entender que el resultado tiene que grabarse sobre el archivo "*central.cpp*", y un intento de sobreescribir este archivo puede llevarnos a perder el código fuente del programa.

La compilación y ejecución, por tanto, podrían realizarse con las siguientes líneas:

```
prompt> g++ -o central central.cpp
prompt> ./central
Introduzca un círculo en formato radio-(x,y): 3-(0,0)
Introduzca otro círculo: 4-(5,0)
El círculo que pasa por los dos centros es: 2.5-(2.5,0)
```

Como puede ver, en la primera hemos compilado y obtenido un ejecutable válido, ya que no se ha generado ningún mensaje de error. En la segunda, hemos lanzado el programa indicando el nombre del archivo ejecutable e indicando que se encuentra en el directorio actual. En la tercera y cuarta hemos introducido dos círculos, y en la última línea nos ha escrito el resultado, un círculo de radio 2.5 centrado en (2.5,0).

> ***Ejercicio** 1: Compilar y ejecutar*
>
> Complete el programa "*central.cpp*" y ejecútelo para comprobar su correcto funcionamiento.

2. Módulos y compilación separada

En el problema que hemos resuelto hemos creado dos nuevos tipos de datos, *Punto* y *Circulo*, así como una serie de operaciones para ellos. Con esas utilidades, resulta más sencillo resolver problemas que requieran de puntos y círculos. Por ejemplo, si ahora queremos crear un nuevo programa que pregunte por un círculo y que escriba el área correspondiente, no tenemos más que crear un nuevo *main,* donde se use el tipo *Circulo* y sus operaciones. Sin embargo, resulta incómodo tener que crear un archivo "*area.cpp*", donde copiar/pegar todo el código de puntos y círculos, para finalmente añadirle una función "*main*" que resuelve este problema.

La creación de una solución basada en módulos independientes, que se compilan de forma separada, nos facilita en gran medida la reutilización de dichos módulos en nuevos programas. En nuestra solución anterior hemos obtenido:

1. Una definición de "*Punto*", y una serie de operaciones con este tipo. Este conjunto de utilidades se puede considerar como una unidad, un módulo que compone nuestra solución.
2. La definición de "*Circulo*", y las operaciones con este tipo. Este módulo hace uso del anterior para definir el centro.
3. Un módulo para calcular el círculo central. Este módulo usa los dos anteriores en una función "*main"* que resuelve un problema muy concreto.

Si hacemos que los tres módulos se escriban de forma independiente, haremos más sencillo crear programas que trabajen con puntos y círculos. Por ejemplo, si ahora queremos obtener el programa "*area*", sólo es necesario crear un "*main"* para este programa e indicar que el programa lo componen este módulo junto con los dos anteriores. Si queremos obtener un programa que nos indique la distancia entre dos puntos, podemos crear una función "*main*" en un archivo "*distancia.cpp*" e indicar que el programa está compuesto por este módulo junto con el módulo "*Punto*".

2.1. Separación en archivos

Para generar los distintos módulos de forma independiente, crearemos distintos archivos fuente (".cpp") para separar cada una de las partes. Concretamente:

1. Fichero "*punto.cpp*". En este archivo se incluye todo lo relacionado con puntos. Así, debe incluir la estructura "*Punto*", junto con las operaciones correspondientes.
2. Fichero "*circulo.cpp*". En este archivo se incluye todo lo relacionado con circulos. Así, debe incluir la estructura "*Circulo*", junto con las operaciones correspondientes.
3. Fichero "*central.cpp*". En este archivo se incluye la función main que implementa el algoritmo de cálculo del círculo central.

Para que esta división sea correcta, será necesario crear dos archivos cabecera para los dos primeros módulos:

1. Fichero "*punto.h*". Este archivo contiene la definición del tipo "*Punto*" y todas las cabeceras de las funciones del módulo. Note que el archivo "*punto.cpp*" no contiene directamente esta estructura, ya que en realidad lo que hace es incluir "*punto.h*". Cuando queramos que un programa use el tipo "Punto" y sus operaciones, no tendremos más que incluir (con "*#include*") este archivo cabecera.
2. Fichero "*circulo.h*". Este archivo contiene la definición del tipo "*Circulo*" y todas las cabeceras de las funciones del módulo. Al igual que el anterior, el "*cpp*" no contiene directamente la estructura, sino que la incluye desde este archivo cabecera. Observe que este archivo siempre va a requerir del tipo "*Punto*" antes que él, ya que es necesario para definir el tipo "*Circulo*". Por tanto, habrá una línea "*#include*" para incluir el archivo "*punto.h*" antes de definir "*Circulo*".

Como puede observar, en el archivo cabecera incluimos todo el código necesario para poder usar las herramientas que "*exporta*" un módulo. Si nuestro programa usa puntos y círculos, y queremos usar los dos módulos, tendremos que incluir ambos archivos cabecera.

Los detalles de cómo obtener cada uno de estos ficheros se pueden consultar, por ejemplo, en el capítulo 12 de [GAR06a].

> ***Ejercicio** 2: Separar archivos*
>
> A partir del programa "*central.cpp*", cree el conjunto de 5 ficheros que componen el resultado de separar los tres módulos.

2.2. Compilación y ejecución

Una vez disponemos de los 5 archivos, podemos realizar la compilación para obtener el ejecutable. Una solución muy simple es la siguiente:

```
prompt> g++ -o central punto.cpp circulo.cpp central.cpp
```

En este caso, hemos escrito una línea similar a cuando teníamos un único archivo, pero especificando los tres. El proceso para obtener el archivo ejecutable realmente es más complejo, ya que es necesario:

1. Compilar cada uno de los tres archivos. Es decir, obtener un fichero objeto (traducción) desde cada uno de los archivos fuente (*cpp*).
2. Enlazar los tres ficheros objeto obtenido en la etapa anterior para crear el ejecutable. En este caso no hay que traducir, sino enlazar. Por ejemplo, en el programa central se llama a la función de lectura de un punto, por lo que en el ejecutable este punto de llamada debe estar "conectado" al punto donde está definida dicha función.

Como resultado podemos obtener distinto tipos de error: errores de compilación en un archivo (nos indicará el archivo y el punto donde encuentra algún problema) o errores de enlazado (nos indicará la función o símbolo que no encuentra o resuelve). Si no indica nada, es que ha obtenido el ejecutable sin problemas.

La compilación y enlazado directo se debe a que el programa "*g++*" nos facilita la operación

al interpretar el tipo de archivos que le damos como entrada. Así, como queremos obtener un ejecutable y le damos tres archivos fuente, internamente realiza las tres traducciones a archivos objeto, así como la llamada al enlazador para obtener el resultado final.

Nosotros estamos interesados en detallar cada uno de los pasos que se deben realizar, y por lo tanto, podemos realizar la misma operación paso a paso:

```
prompt> g++ -c -o central.o central.cpp
prompt> g++ -c -o punto.o punto.cpp
prompt> g++ -c -o circulo.o circulo.cpp
prompt> g++ -o central punto.o circulo.o central.o
```

Ahora hemos obtenido 4 archivos nuevos:

1. Tres archivos objeto (".o") con las tres primeras líneas.
2. Un ejecutable, en la última línea.

En estas líneas podemos distinguir:

- En la compilación se indica el archivo fuente y el objeto que vamos a obtener. Como sabemos, la opción "*-o*" precede el nombre del archivo resultado. Le asignamos esa extensión para distinguirlo como archivo objeto.
- En la compilación se indica la opción "*-c*". Con esta opción el compilador sabe que se tiene que limitar a compilar, es decir, traducir en un archivo objeto que no es ejecutable.
- En la última línea, no existe la opción "*-c*". Cuando no existe, *g++* entiende que la intención del usuario es obtener un ejecutable, por lo que realizará lo necesario para obtener el ejecutable. En nuestro caso, simplemente enlazar.
- Se compilan archivos "*cpp*". Como era de esperar, los archivos cabecera ("*.h*") sólo existen para ser insertados en los archivos "*cpp*" a través de la directiva "*#include*". Podríamos decir que no son más que "trozos" de código c++ que se insertan en los archivos "*cpp*" donde se compilarán como parte de un fichero más complejo.

> ***Ejercicio** 3: Compilar, enlazar y ejecutar*
>
> Realice la compilación, enlazado del programa a partir de los 3 archivos fuente que hemos indicado. Ejecute para comprobar su correcto funcionamiento.

2.3. Bibliotecas

El problema que hemos resuelto ha dado lugar a dos módulos reutilizables: *Punto* y *Circulo*. En la práctica, podemos crear programas que usen las herramientas contenidas en uno o en los dos módulos. Por ejemplo:

1. Distancia entre puntos. Para crear un programa que obtenga la distancia entre dos puntos, se puede obtener el ejecutable creando un archivo con un "*main*" y enlazándolo con el módulo "*Punto*".
2. Área. Para crear un programa que lee un círculo y obtiene el área del círculo, podemos crear el ejecutable con un "*main*" que se enlaza con los dos módulos, "*Punto*" y "*Circulo*".

En el primer caso habrá que enlazar con un solo módulo, y en el segundo, con los dos. Ahora bien, en la práctica es posible crear módulos reutilizables mucho más numerosos y complejos.

Por ejemplo, imagine que creamos un conjunto de módulos para operar con formas del espacio 2D (puntos, círculos, rectángulos, triángulos, etc.). Podemos obtener un número bastante grande de módulos, que tendremos que gestionar para poder crear programas que usen alguno o varios de dichos módulos.

La gestión de un grupo de módulos relacionados se puede simplificar por medio de las bibliotecas[1]. Una biblioteca no es más que un grupo de módulos compilados y empaquetados en un mismo archivo. Podríamos decir que es un contenedor de archivos objeto ("*.o*").

Cuando usamos bibliotecas no será necesario indicar todos y cada uno de los módulos objeto que hacen falta para obtener el ejecutable. En lugar de eso, basta con indicar el nombre de la biblioteca, y el enlazador se encarga de extraer y añadir los módulos que hagan falta para el programa ejecutable. Observe que no es necesario añadirlos todos, sino que sólo se usarán los que el programa requiera.

Así, si tenemos una biblioteca con 100 archivos objeto empaquetados, es posible que el programa use las herramientas de uno solo de ellos, y por tanto, el enlazador sólo extraiga y añada ese módulo. Es más, incluso si indicamos que enlace con una biblioteca, podría no extraer ninguno si no fuera realmente necesaria.

Para crear archivos biblioteca será necesario usar la orden "*ar*" (de "*archive*"). La forma en que vamos a usar esta orden se puede indicar como sigue:

```
ar <operación> <biblioteca> [<archivos>]
```

donde podemos distinguir tres partes:

1. Operación. En general, es una letra que indica lo que queremos realizar. Por ejemplo, podemos añadir archivos, reemplazar, consultar, etc.
2. Biblioteca. Es el nombre del archivo biblioteca con el que queremos trabajar.
3. Archivos. Podemos indicar cero o más archivos objeto con los que realizar la operación.

Aunque existen múltiples posibilidades, nosotros vamos a usar esta orden de una manera muy concreta y simple, de forma que todo lo que vamos a necesitar se puede realizar con una orden:

1. La operación la especificaremos siempre como "*rvs*". La operación, en concreto, es "r", es decir, insertar módulos con reemplazo. Las letras "vs" se indican para obtener información de lo que se hace -la primera- y para que se añada o actualice un índice con los contenidos del archivo -la segunda-.
2. La biblioteca será el nombre del archivo con el que trabajar. Si el nombre existe, la operación de inserción se encarga, además, de crear el archivo.
3. El módulo o módulos a incluir en la biblioteca. Si no están ya, se insertarán como nuevos, y si ya existen, se reemplazan.

En nuestro ejemplo de puntos y círculos, hemos creado dos módulos con los que enlazar nuestros programas. Podemos crear una biblioteca con la siguiente línea:

```
prompt> ar rvs libformas.a punto.o circulo.o
```

Observe el nombre que hemos creado para la biblioteca. El primer lugar, tiene una extensión "*.a*" (de "*archive*"). Por otro lado, empieza por "*lib*" (de "*library*"). Todos los nombres tendrán esta estructura:

```
lib<nombre>.a
```

Por tanto, la biblioteca que hemos creado se llama "*formas*". Es interesante destacar que si ahora modificamos uno de los archivos objeto, por ejemplo, porque hemos cambiado el "*cpp*" y lo recompilamos, podemos ejecutar exactamente la misma orden para obtener la biblioteca actualizada.

---

1 *En inglés "library". Está muy extendido el uso de la palabra "librería" en lugar de biblioteca debido a una mala traducción del inglés.*

2.3.1. Enlazar con bibliotecas

La forma directa y simple para comprobar que nuestra nueva biblioteca funciona correctamente y nos permite obtener el ejecutable deseado es indicando el nombre de ésta en lugar de los archivos objeto. Es decir, enlazar con la siguiente orden:

```
prompt> g++ -o central central.o libformas.a
```

Con lo que obtendríamos exactamente el mismo ejecutable. Sin embargo, no es la forma habitual de usar las bibliotecas. Normalmente se usa la opción "-l" de "g++" para indicar una biblioteca con la que enlazar. La línea podría ser la siguiente:

```
prompt> g++ -o central central.o -lformas
```

Observe que la opción va seguida del nombre de la biblioteca (no del archivo). Si ejecuta esta línea es probable que obtenga un error de enlazado, ya que nos indica que no encuentra la biblioteca "*formas*". El problema es que cuando decimos de enlazar con una biblioteca, el enlazador tiene un conjunto de directorios muy concretos donde encontrar las bibliotecas. Lógicamente, en principio, sólo contiene algunos directorios del sistema, donde se encuentran las bibliotecas que se hayan instalado, y no el directorio actual.

La solución al problema es sencilla, pues no tenemos más que incluir un nuevo directorio donde buscar bibliotecas. La biblioteca "*formas*" está creada en el directorio donde estamos trabajando, así que si incluimos el directorio actual para buscarla, el enlazador podrá obtener el resultado que queremos. El directorio donde trabajamos se puede indicar con ".", así que añadimos este directorio por medio de la opción "*-L*" (en mayúscula). En concreto:

```
prompt> g++ -o central central.o -L. -lformas
```

> ***Ejercicio** 4: Crear y usar una biblioteca*
>
> Use la orde "*ar*" para crear una biblioteca "*formas*" y vuelva a generar el ejecutable enlazando con ella.

Finalmente, es importante enfatizar el resultado que hemos obtenido. La biblioteca no es más que una colección de archivos objeto que se puede enlazar para obtener programas ejecutables.

Si queremos obtener nuevos programas, no sólo necesitamos la biblioteca, sino los archivos cabecera que deberán incluirse para acceder a las utilidades que ofrecen. Por ejemplo, aunque tengamos el archivo "*libformas.a*", nunca podremos obtener el archivo "*central.o*" si no disponemos de los dos archivos cabecera ("*punto.h*" y "*circulo.h*"). Por esta razón, cuando se distribuye una biblioteca para que los programadores desarrollen nuevos programas, seguramente tendrá un grupo de archivos que incluye archivos cabecera.

2.3.2. Orden de bibliotecas

Inicialmente, hemos presentado la orden "*g++*" indicando que el orden de los parámetros no es relevante, pues podemos cambiarlo obteniendo el mismo resultado. Sin embargo, las bibliotecas que aparecen en una línea deben estar correctamente ordenadas.

Básicamente, el enlazador pasa por cada una de las bibliotecas de izquierda a derecha, una sola vez, extrayendo y añadiendo lo que se necesite. Para decidir lo que necesita, debe revisar el conjunto de símbolos sin resolver que están pendientes.

Por ejemplo, si en el programa tenemos una llamada a una función "*Distancia*", al pasar por las bibliotecas consulta si esta función está incluida en algún módulo objeto, en cuyo caso es añadido al ejecutable. Ahora bien, una vez añadida esta función, se pueden haber creado nuevos símbolos que resolver (que esta función llama). Estos nuevos símbolos se buscarán en las siguientes bibliotecas, y no en las ya revisadas.

3. Gestión del proyecto con make

En las secciones anteriores se han presentado las órdenes necesarias para compilar un proyecto con múltiples archivos, incluyendo una biblioteca y un ejecutable. Estas órdenes se ejecutan una vez están terminados todos los archivos fuente. Sin embargo, en la práctica, es normal que se estén desarrollando los archivos fuente mientras se están recompilando los programas, ya sea para corregir errores de compilación, para modificar el programa, para ampliarlo, etc.

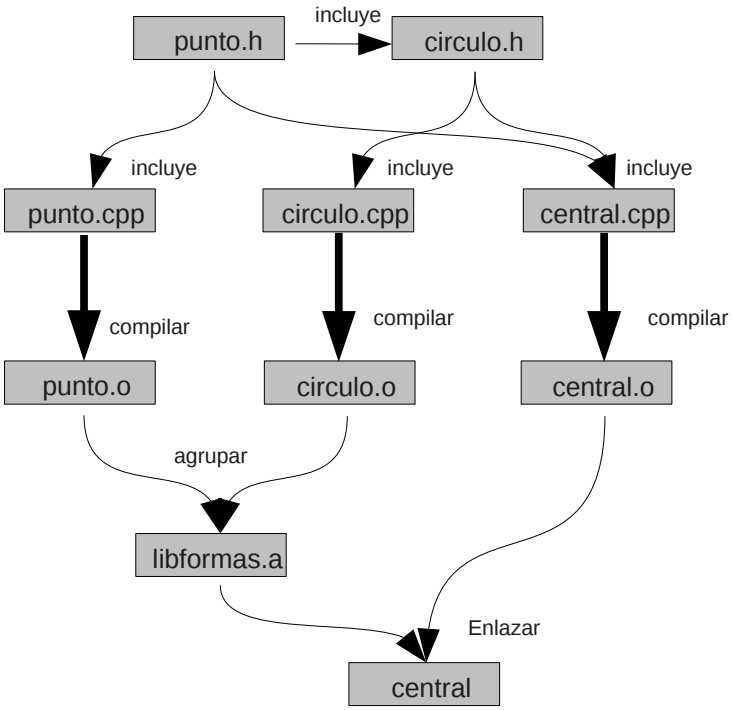
La separación de la solución en módulos independientes facilita y hace más eficiente esta forma de trabajo. Por ejemplo, si desarrollamos el módulo "*punto.cpp*", y lo compilamos, ya no será necesario volver a compilarlo mientras no lo modifiquemos. Así, si una vez que tenemos un ejecutable, modificamos una línea del archivo "*central.cpp*", sólo será necesario volver a compilar este módulo y enlazar el resultado con la biblioteca para actualizar el ejecutable.

El problema surge cuando realizamos un cambio en una parte del proyecto que afecta a más módulos, especialmente si tenemos un proyecto mucho más complejo. En este caso, puede ser muy complicado determinar los módulos que tienen que volverse a compilar, lo que nos puede llevar a tener que recompilarlo todo para asegurarnos de que se actualiza correctamente.

La orden "*make*" nos permite gestionar todos los archivos de un proyecto y ayudarnos a realizar las actualizaciones necesarias para generar los archivos ejecutables.

3.1. Dependencias entre módulos

Si analizamos los módulos que hemos generado en las secciones anteriores podemos obtener el siguiente resultado:



En este gráfico de dependencias se refleja claramente cómo podemos llegar desde los archivos fuente hasta el archivo ejecutable "*central*". Por tanto, conociendo estas dependencias, junto con las órdenes de compilación que hemos estudiado, podemos ser capaces de regenerar el ejecutable cuando realicemos alguna modificación. Observe que:

- Las etiquetas incluye corresponden a directivas "*#include*" del código. El archivo "*central.cpp*" incluye los dos cabecera, aunque podría haberse creado incluyendo sólo el archivo "*circulo.h*". Si se hubiera incluido sólo éste, podríamos eliminar una línea de inclusión desde "*punto.h*" al "*central.cpp*". Sin embargo, observe que todavía se podría observar que el archivo "*punto.h*" se incluye indirectamente al incluir "*circulo.h*".
- Las etiquetas "*compilar"*, "*agrupar"* y "*enlazar"* nos indican que esos módulos se obtienen a partir de los anteriores con los comandos "*g++ -c*", "*ar*", y "*g++*" respectivamente.

A partir de este gráfico podemos determinan el conjunto de acciones necesarias cuando se modifica uno de los 5 archivos fuente. Por ejemplo, si modificamos "*central.cpp*", podemos ver que será necesario:

1. Regenerar "*central.o*", ya que ahora el programa es distinto. Hay que volver a compilar.
2. Regenerar el ejecutable "*central*". Al haber modificado el módulo objeto, hay que volver a obtener el ejecutable. Hay que volver a enlazar.

En este supuesto, no hemos tenido que modificar ningún otro módulo. En concreto, la biblioteca que se usa en el enlazado no se ha cambiado, ya que el código que se incluye para compilarla es exactamente el mismo.

Un ejemplo más complejo ocurre al modificar "*circulo.h*". Como sabemos, este módulo no se compila directamente, sino que se incluye con la directiva "*#include*". Eso significa que los dos módulos compilables "*circulo.cpp*" y "*central.cpp*", a pesar de no haberse modificado directamente, se ven afectados. Cuando se compilan, se inserta el código que haya escrito en el fichero "*circulo.h*", y por tanto, si modificamos este archivo, estamos modificando el código que llega al compilador desde los "*cpp*". Será necesario:

1. Regenerar "*central.o*". Hay que volver a compilar.
2. Regenerar "*circulo.o*". Hay que volver a compilar.
3. Regenerar "*libformas.a*". El módulo "*circulo.o*" que tenía antes ya no es válido, pues se ha obtenido uno nuevo. Hay que reemplazar ese módulo.
4. Regenerar el ejecutable "*central*". Hay que volver a enlazar.

Podemos hacer nuevos supuestos, que implicarán las correspondientes actualizaciones. Por ejemplo, fíjese que si modificamos el archivo "*punto.h*", será necesario rehacerlo todo.

La orden "*make*" nos permitirá gestionar todas estas dependencias de forma automática. En lugar de estudiar los pasos que son necesarios para rehacer el ejecutable, usaremos la orden "*make*" para que estudie las dependencias que "han fallado", y lance automáticamente cada uno de los pasos para regenerar el archivo ejecutable.

3.2. Archivo makefile

La orden "*make*" necesita conocer el conjunto de módulos, las dependencias entre ellos, y la forma de generarlos para poder gestionar un proyecto de programación. Un archivo "*makefile"* es un archivo de texto que contiene esta información en un formato que la orden "*make*" sabe interpretar.

Este tipo de archivo lo nombramos así porque la mayoría de las veces será un archivo de texto que tenga exactamente este nombre (o con la primera letra mayúscula). En general, el nombre de un archivo que contiene información para "make" no tiene por qué llamarse así, de hecho puede ser cualquier nombre, aunque nosotros siempre usaremos este para facilitar su uso.

3.2.1. Reglas

El contenido del archivo "*makefile*" permite representar la información que hemos mostrado en el esquema anterior, es decir, permite indicar cuáles son los módulos, cuáles son las dependencias, y cuáles son los comandos necesarios para regenerarlos.

Aunque el gráfico parece muy complejo, al incluir múltiples dependencias directas e indirectas, en la práctica es muy simple especificarlo en forma de texto, ya que lo que hay que incluir en el archivo "makefile" es cada uno de los componentes del proyecto. Concretamente, habrá que especificar:

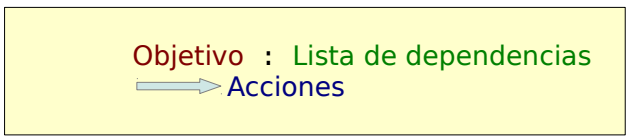
1. El archivo "*punto.o*" se debe crear con una orden "*g++ -c*" siempre y cuando se hayan modificado los archivos "*punto.h*" o "*punto.cpp*".
2. El archivo "*circulo.o*" se debe crear con una orden "*g++ -c*" siempre y cuando se hayan modificado los archivos "*punto.h*", "*circulo.h*" o "*circulo.cpp*".
3. El archivo "*central.o*" se debe crear con una orden "*g++ -c*" siempre y cuando se hayan modificado los archivos "*punto.h*", "*circulo.h*" o "*central.cpp*".
4. El archivo "*libformas.a*" se debe crear con una orden "*ar*" siempre y cuando se hayan modificado los archivos "*punto.o*" o "*circulo.o*".
5. El archivo "*central*" se debe crear con una orden "*g++*" siempre y cuando se hayan modificado los archivos "*central.o*" o "*libformas.a*".

Observe que aunque los items son independientes, es fácil ver que están interrelacionados. Por ejemplo, podemos deducir que si modificamos el archivo "*central.cpp*", el punto 3 indica que será necesario crear el archivo "*central.o*", pero igualmente, al volver modificar éste, vemos que el punto 5 nos indica que debemos generar el archivo "*central*".

Esta información es la que vamos a incluir, con un formato muy concreto, en un archivo "*makefile*". La orden "*make*" es la encargada de interpretar estas instrucciones para lanzar todas las órdenes necesarias.

¿Cómo puede saber que un archivo se ha modificado y hay que volver a regenerar un módulo? Simplemente, comprobando las fechas de último acceso al archivo. En el disco, junto a los ficheros, se almacena la fecha y hora de última modificación. Si comprobamos las fechas de "*central.o*" y "*central.cpp*", podemos determinar que "*central.o*" hay que regenerarlo si vemos que su fecha es anterior a la de "*central.cpp*".

La estructura básica para representar esta información es una "*regla*":



En donde:

- Objetivo: Indica lo que queremos construir. Por ejemplo, el módulo "*libformas.a*".
- Lista de dependencias: Esto es una lista de items de los que depende la construcción del objetivo de la regla. 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. Por ejemplo, "*libformas.a*" es un objetivo que depende de "*punto.o*" y "*circulo.o*".
- 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. Por ejemplo, el objetivo "*libformas.a*" se consigue lanzando una orden "*ar rvs libformas.a punto.o circulo.o*".

Por tanto, en nuestro problema, usaremos una regla para codificar cada una de las 5 ficheros que queremos controlar. Además, aunque se escriban como reglas independientes, la orden "*make*" sabe relacionar unas con otras, ya que las dependencias de unas reglas son los objetivos de otras.

En nuestro ejemplo podemos escribir las siguientes reglas para representar ese gráfico:

```
punto.o  :  punto.h punto.cpp
        g++ -c -o punto.o punto.cpp
circulo.o  :  circulo.h circulo.cpp
        g++ -c -o circulo.o circulo.cpp
central.o  :  punto.h circulo.h central.cpp
        g++ -c -o central.o central.cpp
central  :  central.o libformas.a
        g++ -o central central.o -L. -lformas
libformas.a  :  punto.o circulo.o
        ar rvs libformas.a punto.o circulo.o
```

Este conjunto de reglas se pueden almacenar en un archivo de texto con nombre "*Makefile*", en el mismo directorio donde tenemos los archivos fuente del programa. Después de ello, podemos procesarlo con la siguiente orden:

```
prompt> make central
```

En la que se ha indicado a "*make*" que intente obtener el objetivo "*central*". No es necesario indicarle el archivo con las reglas, ya que al no especificar nada, buscará automáticamente un archivo con el nombre "*Makefile*"[2].

Respecto a la sintaxis usada para escribir estas reglas en el archivo "*makefile*", tenga en cuenta que:

- El orden de las reglas no afecta al resultado.
- Las acciones (u órdenes) que se incluyen en cada regla deben estar precedidas por un tabulador.

El hecho de añadir un tabulador a las acciones es fundamental para que funcione correctamente. Una regla puede tener un número indeterminado de acciones a realizar (una detrás de otra) y la forma de saber si hay más acciones es comprobando si la siguiente línea comienza con un tabulador.

Finalmente, aunque el orden de las reglas no afecte al resultado, es habitual situar la regla con el objetivo final como primera regla. Así, en nuestro ejemplo, la regla del objetivo "*central*" se situaría en primer lugar. Con este orden el uso de "*make*" se hace aún más simple, ya que si llamamos a "*make*" sin indicar ningún objetivo, entenderá que nuestro objetivo es el de la primera regla.

> ***Ejercicio*** *5: Archivo "makefile" simple*
>
> Edite un archivo "*Makefile*" en el directorio de trabajo y compruebe que funciona correctamente. Para ello, haga alguna modificación en un archivo fuente y vuelva a lanzar la orden "*make*".

3.2.2. Macros

En un archivo "*makefile*" se pueden incluir macros como identificadores que nos sirven para parametrizar el archivo. Su sintaxis en la siguiente:

```
<NOMBRE> = <texto correspondiente>
```

donde

- <NOMBRE>: es el nombre de la macro, y corresponde a un identificador (letras, dígitos y '_') que normalmente se escribe con todas las letras en mayúscula.
- <texto correspondiente>: es el texto por el que sustituir la macro.

2 *Si hubiéramos puesto un nombre distinto, tendríamos que haber indicado ese nombre con la opción -f.*

Para usar una macro, simplemente escribimos el nombre entre paréntesis (o llaves) precedido de carácter "$".

Existen una lista bastante grande de nombres que se usan habitualmente para parametrizar algunos valores en un archivo "*makefile*". Algunas de ellas son:

- `AR:` Programa para mantener las bibliotecas.
- `CXX:` Programa que se usa para compilar código c++.
- `CXXFLAGS:` Opciones (*flags*) adicionales para añadir al compilador de c++.
- `LDFLAGS:` Opciones (*flags*) adicionales para añadir cuando se invoca al enlazador.

Podemos definir y usar estas variables en nuestro archivo "*makefile*". Por ejemplo, nuestro archivo "*makefile*" podría comenzar como sigue:

```
AR= ar
CXX = g++
CXXFLAGS= -Wall -g
LDFLAGS= -L. -lformas


punto.o  :  punto.h punto.cpp
        $(CXX) -c $(CXXFLAGS) -o punto.o punto.cpp
```

De esta forma, es muy simple realizar un pequeño cambio que afecte múltiples reglas. Por ejemplo, según este ejemplo hemos añadido la opción "*-g*" en la compilación, es decir, hemos indicado que queremos obtener un código preparado para ser procesado en el depurador. Cuando tengamos terminado todo el programa, y no sea necesario depurar, podemos cambiar esa opción por "*-O*", es decir, generar un código que no se puede usar con el depurador, pero que es más eficiente.

> ***Ejercicio** 6: Incluir macros habituales.*
>
> Incluya las macros que se han comentado en su fichero "*makefile*" y adapte las reglas para que haga uso de ellas.

3.2.3. Múltiples objetivos finales

El proyecto que hemos resuelto nos permite obtener un archivo ejecutable ("*central*") a partir de un conjunto de fuentes. Al escribir la orden "*make*" en el terminal obtenemos el ejecutable correspondiente, ya que la regla de este objetivo está en primer lugar. Recordemos que esta orden, sin parámetros, hace que se actualice el primer objetivo, es decir, el de la primera regla.

Si el proyecto es más complejo, es probable que tengamos varios programas a generar. Por ejemplo, imagine que además del ejecutable "*central*" deseamos también un ejecutable "*area*", que corresponde a un archivo "*area.cpp*" que se compila y enlaza con nuestra biblioteca. Al tener dos reglas, una para "*central*" y otra para "*area*", sólo una de ellas puede estar en primer lugar, por lo que escribir "*make*" sin nada más nos llevaría a obtener sólo uno de los objetivos. Por supuesto, siempre podemos hacer "*make central*", y "*make area*", aunque no es la solución más cómoda, especialmente si hay muchos objetivos a alcanzar.

Para resolver el problema podemos usar un objetivo simbólico, o ficticio, que sólo nos sirve para poder generar ambos ejecutables, sin ser realmente un archivo a generar. En concreto, escribimos una primera regla vacía (sin acciones), con un nombre de objetivo cualquiera (no se va a generar el archivo) y con unas dependencias que corresponden a todos los objetivos que deseamos generar.

Por ejemplo, a continuación mostramos el archivo "makefile" modificado, incluyendo una primera regla ficticia:

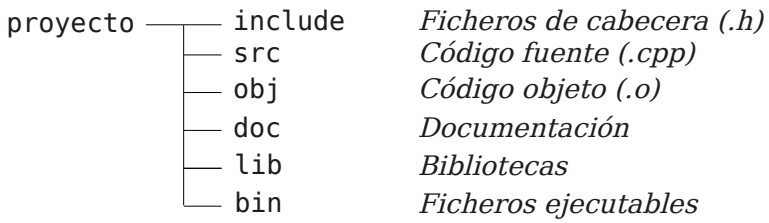
```
all  :  central area

central  :  central.o libformas.a
        g++ -o central central.o -L. -lformas

area  :  area.o libformas.a
        g++ -o area area.o -L. -lformas
```

donde vemos que también se presentan las reglas para “central” y “area”, que son los dos ejecutables finales deseados. Se ha puesto un objetivo (“*all*”) que depende de “*central*” y “*area*”, como primer objetivo. La ejecución de “*make*” sin parámetros lanza la comprobación de la primera regla, por lo que es necesario comprobar las dependencias, lo que provoca la generación de los dos ejecutables. Lógicamente, al terminar de comprobar las dependencias, el programa termina al no encontrar acciones que realizar.

Finalmente, observe que hemos llamado al objetivo “*all*”, aunque podría haber sido cualquier otro. Este nombre suele ser habitual, ya que expresa la intención de esta regla, es decir, que se ha creado para poder generar “*todo*”.

3.2.4. Distribución en directorios

Normalmente no se incluyen todos los archivos del proyecto en un mismo directorio, sino que se dividen en distintos directorios, separando fuentes de archivos generados, y facilitando así el manejo de toda la información. Por ejemplo, se pueden dividir los archivos según el siguiente esquema:



de forma que el archivo “*Makefile*” estaría en el directorio “proyecto”, y los demás archivos distribuidos en cada directorio.

Con esta distribución ya no serían válidas las reglas que hemos escrito para gestionar el proyecto, ya que buscaría los ficheros en el directorio actual. Para resolverlo, sería necesario añadir a cada fichero el directorio donde se encuentra. Por ejemplo, la siguiente regla sí tendría en cuenta los directorios donde se encuentran los archivos:

```
obj/punto.o  :  include/punto.h src/punto.cpp
        $(CXX) -c $(CXXFLAGS) -o obj/punto.o src/punto.cpp
```

Además, si queremos que los directorios donde se encuentran los archivos puedan cambiarse fácilmente, podemos parametrizar el nombre de estos directorios en un grupo de macros al principio del archivo.

> ***Ejercicio*** *7: Destribuir en directorios.*
>
> Añada macros “*INC,SRC,OBJ,LIB,BIN*” para parametrizar los directorios donde se encuentran los archivos y modifique las reglas para tenerlas en cuenta.

Observe que la simple sustitución no es suficiente para que las órdenes de compilación funcionen correctamente. Este problema se debe a que los "*#include*" de nuestros archivos han dejado de funcionar.

Antes, el compilador encontraba los archivos que se incluían, ya que se encontraban en el mismo directorio, pero ahora los archivos "*.cpp*" no están en el mismo lugar que los "*.h*". Para resolverlo, añada la opción "*-I*" al compilador. Esta opción va seguida de un directorio (similar a la opción "*-L*" para bibliotecas) donde encontrar archivos cabecera. Bastará con añadirla a la lista de opciones de "*CXXFLAGS*", indicando la opción "*-I*" seguida del directorio guardado en la macro "*INC*".

3.2.5. 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 definitivamente el proyecto, y generados los ejecutables, no serán necesarios los códigos objeto ni las bibliotecas, por lo que podemos borrarlos.

Para facilitar esta tarea se suelen incorporar algunas reglas que hacen estas tareas de limpieza. Es frecuente considerar dos niveles de limpieza del proyecto. Un primer nivel que limpia ficheros intermedios, pero deja las aplicaciones finales que hayan sido generadas:

```
clean:
    echo "Limpiando ..."
    rm $(OBJ)/*.o $(LIB)/lib*.a
```

De esta forma, tras acabar de programar y depurar el proyecto podemos quedarnos únicamente con el código fuente 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:

```
mrproper: clean
    rm $(BIN)/central
```

De esta forma lo que conseguimos es que se aplique la regla `clean` y que, a continuación, se apliquen las acciones de esta regla. De esta manera obtendremos el proyecto en un estado de "únicamente fuentes".

Esta situación es la ideal para empaquetar el proyecto y llevarlo a otro lugar para generarlo, es decir, para distribuir el proyecto para que sea compilado en otros sistemas. Por ejemplo, en nuestro proyecto de círculo central, podemos ponernos en el nivel de la carpeta "proyecto" y escribir:

```
prompt> tar zcvf central.tgz proyecto
```

para generar un nuevo archivo "*central.tgz*" que contiene todos los directorios y archivos que componen el proyecto. Llevando este archivo a otro sistema, y ejecutando

```
prompt> tar zxvf central.tgz
prompt> cd proyecto
prompt> make
```

Nos permite primero desplegar todo el contenido de archivos empaquetado y segundo regenerar los ejecutables.

3.2.6. Reglas implícitas

En las secciones anteriores hemos indicado explícitamente una serie de reglas que nos permiten generar nuestros ejecutables. La orden "*make*" puede, además, usar reglas implícitas, es decir, que no están escritas con todo detalle.

Como hemos visto, la forma en que se generan algunos ficheros responde a un patrón. Por ejemplo, un arhivo "*.o*" se obtiene desde un "*.cpp*" lanzando el compilador, seguido de las opciones de compilación, el archivo destino y el archivo fuente.

Por tanto, la orden "*make*" puede tener en cuenta estas reglas implícitas en caso de querer obtener un arhivo "*.o*" a a partir de un "*.cpp*" y no disponer de la regla explícita que indique cómo hacerlo. No sólo contiene un conjunto de reglas implícitas predefinidas, sino que podríamos escribir nosotros las nuestras para que se adaptaran a nuestro proyecto.

Las reglas implícitas son una herramienta muy útil, especialmente en proyectos donde haya un número muy alto de objetivos a obtener. Por ejemplo, imagine que tenemos un proyecto con 50 archivos "*cpp*" que compilar. Sería muy tedioso tener que escribir todas las reglas. Es más fácil indicar un patrón y que lo aplique a todos los archivos.

Dado que nuestros proyectos son relativamente simples, no haremos uso de este tipo de reglas, sino que escribiremos explícitamente cada una de ellas. A pesar de ello, es importante tener en cuenta su existencia, ya que es posible que escriba un fichero "*makefile*", se olvide de poner alguna regla, y al procesarlo obtener un conjunto de órdenes a las que no encuentra sentido, ya que no las ha escrito, sino que han sido generadas a partir de las reglas implícitas.

Finalmente, es interesante destacar que los nombres de las macros que se han presentado como de uso habitual (*AR*, *CXX*, etc.) son también de interés para las reglas implícitas, ya que son éstos los que se usan en las reglas implícitas para generar la orden concreta.

4. Referencias

- [GAR06a] Garrido, A. "*Fundamentos de programación en C++*". Delta publicaciones, 2006.
- [GAR06b] Garrido, A. Fdez-Valdivia, J. "*Abstracción y estructuras de datos en C++*". Delta publicaciones, 2006.
- [STR02] Stroustrup, B. "*El lenguaje de programación C++*". Edición Especial. Addison Wesley, 2002.
- [MAKE] GNU, "GNU Make". http://www.gnu.org/software/make/manual/make.html