Grado en Ingeniería Informática. Metodología de la Programación.

Práctica 3. Compilación separada. Gestión y uso de bibliotecas. El programa `ar`

Francisco J. Cortijo Bon

Curso 2012-2013

Objetivos

1. Conocer las ventajas de la modularización
2. Saber cómo organizar un proyecto software en distintos ficheros, cómo se relacionan y cómo se gestionan usando un fichero *makefile*.
3. Entender el concepto de *biblioteca* en programación.
4. Conocer el funcionamiento de la orden `ar`.
5. Saber cómo enlazar ficheros de biblioteca.
6. Apender a gestionar y enlazar bibliotecas en ficheros *makefile*.

Índice

1. La modularización del software en C++ **33**
1.1. Introducción . . . 33
1.2. Ventajas de la modularización del software . . . 33
1.3. Cómo dividir un programa en varios ficheros . . . 34
1.4. Organización de los ficheros fuente . . . 34

2. Bibliotecas **38**
2.1. Tipos de bibliotecas . . . 39
2.2. Estructura de una biblioteca . . . 39

3. El programa `ar` **42**

4. `g++`, `make` y `ar` trabajando conjuntamente **43**
4.1. Creación de la biblioteca . . . 43
4.2. Enlazar con una biblioteca . . . 44

5. Ejercicios **45**

1. La modularización del software en C++

1.1. Introducción

Cuando se escriben programas medianos o grandes resulta sumamente recomendable (por no decir *obligado*) dividirlos en diferentes módulos fuente. Este enfoque proporciona muchas e importantes ventajas, aunque complica en cierta medida la tarea de compilación.

Básicamente, lo que se hará será dividir nuestro programa fuente en varios ficheros. Estos ficheros se compilarán por separado, obteniendo diferentes ficheros objeto. Una vez obtenidos, los módulos objeto se pueden, si se desea, reunir para formar bibliotecas. Para obtener el programa ejecutable, se enlazará el módulo objeto que contiene la función `main()` con varios módulos objeto y/o bibliotecas.

1.2. Ventajas de la modularización del software

1. Los módulos contendrán de forma natural conjuntos de funciones relacionadas desde un punto de vista lógico.
2. Resulta fácil aplicar un enfoque orientado a objetos. Cada objeto (tipo de dato abstracto) se agrupa en un módulo junto con las operaciones del tipo definido.
3. El programa puede ser desarrollado por un equipo de programadores de forma cómoda. Cada programador puede trabajar en distintos aspectos del programa, localizados en diferentes módulos. pueden reusarse en otros programas, reduciendo el tiempo y coste del desarrollo del software.
4. La compilación de los módulos se puede realizar por separado. Cuando un módulo está validado y compilado no será preciso recompilarlo. Además, cuando haya que modificar el programa, sólo tendremos que recompilar los ficheros fuente alterados por la modificación. La utilidad `make` será muy útil para esta tarea.
5. El ocultamiento de información puede conseguirse con la modularización. El usuario de un módulo objeto o de una biblioteca de módulos desconoce los detalles de implementación de las funciones y objetos definidos en éstos. Mediante el uso de ficheros de cabecera proporcionaremos la interface necesaria para poder usar estas funciones y objetos.

1.3. Cómo dividir un programa en varios ficheros

Cuando se divide un programa en varios ficheros, cada uno de ellos contendrá una o más funciones. Sólo uno de estos ficheros contendrá la función `main()`. Los programadores normalmente comienzan a diseñar un programa dividiendo el problema en subtareas que resulten más manejables. Cada una de estas tareas se implementará como una o más funciones. Normalmente, todas las funciones de una subtarea residen en un fichero fuente.

Por ejemplo, cuando se realice la implementación de tipos de datos abstractos definidos por el programador, lo normal será incluir todas las funciones que acceden al tipo definido en el mismo fichero. Esto supone varias ventajas importantes:

1. La estructura de datos se puede utilizar de forma sencilla en otros programas.
2. Las funciones relacionadas se almacenan juntas.
3. Cualquier cambio posterior en la estructura de datos, requiere que se modifique y recompile el mínimo número de ficheros y sólo los imprescindibles.

Cuando las funciones de un módulo invocan objetos o funciones que se encuentran definidos en otros módulos, necesitan alguna información acerca de cómo realizar estas llamadas. El compilador requiere que se proporcionen declaraciones de funciones y/o objetos definidos en otros módulos. La mejor forma de hacerlo (y, probablemente, la única razonable) es mediante la creación de ficheros de

cabecera (`.h`), que contienen las declaraciones de las funciones definidas en el fichero `.cpp` correspondiente. De esta forma, cuando un módulo requiera invocar funciones definidas en otros módulos, bastará con que se inserte una línea `#include` del fichero de cabecera apropiado.

1.4. Organización de los ficheros fuente

Los ficheros fuente que componen nuestros programas deben estar organizados en un cierto orden. Normalmente será el siguiente:

1. Una primera parte formada por una serie de constantes simbólicas en líneas `#define`, una serie de líneas `#include` para incluir ficheros de cabecera y redefiniciones (`typedef`) de los tipos de datos que se van a tratar.
2. La declaración de variables externas y su inicialización, si es el caso. Se recuerda que, en general, no es recomendable su uso.
3. Una serie de funciones.

El orden de los elementos es importante, ya que en el lenguaje C++, cualquier objeto debe estar declarado antes de que sea usado, y en particular, las funciones deben declararse antes de que se realice cualquier llamada a ellas. Se nos presentan dos posibilidades:

1. **Que la definición de la función se encuentre en el mismo fichero en el que se realiza la llamada**. En este caso,
   a) se sitúa la definición de la función antes de la llamada (ésta es una solución raramente empleada por los programadores),
   b) se puede incluir una linea de declaración (prototipo) al principio del fichero, con lo que la definición se puede situar en cualquier punto del fichero, incluso después de las llamadas a ésta.
2. **Que la definición de la función se encuentre en otro fichero diferente al que contiene la llamada**. En este caso es preciso incluir al principio del mismo una línea de declaración (prototipo) de la función en el fichero que contiene las llamadas a ésta. Normalmente se realiza incluyendo (mediante la directiva de preprocesamiento `#include`) el fichero de cabecera asociado al módulo que contiene la definición de la función.

   Cuando modularizamos nuestros programas, hemos de hacer un uso adecuado de los ficheros de cabecera asociados a los módulos que estamos desarrollando. Además, los ficheros de cabecera también son útiles para contener las declaraciones de tipos, funciones y otros objetos que necesitemos compartir entre varios de nuestros módulos.

   Así pues, a la hora de crear el fichero de cabecera asociado a un módulo debemos tener en cuenta qué objetos del módulo van a ser compartidos o invocados desde otros módulos (**públicos**) y cuáles serán estrictamente **privados** al módulo: en el fichero de cabecera pondremos, únicamente, los públicos.

   Recordar: *En C++, todos los objetos deben estar declarados y definidos antes de ser usados*.

Ejercicio

Copiar el fichero `unico.cpp` en vuestro directorio de trabajo. Generar el ejecutable `unico`.

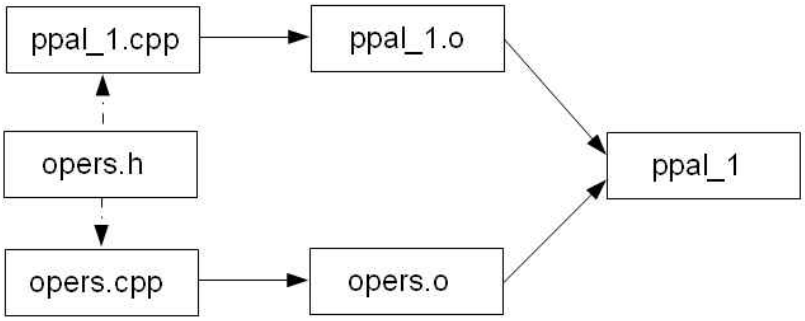

Figura 1: Diagrama de dependencias para construir `ppal_1`

Ejercicio

Organizar el código fuente en distintos ficheros de manera que separaremos la declaración y la definición de las funciones. Guardar cada fichero en la carpeta adecuada.

Los ficheros involucrados y sus dependencias se muestran en la figura 1. Con más detalle:

1. El fichero (`ppal_1.cpp`) contendrá la función `main()`
2. El código asociado a las funciones se organiza en dos ficheros: uno contiene las declaraciones (`opers.h`) y otro las definiciones (`opers.cpp`)

Realizar las siguientes tareas:

1. Generar el objeto `ppal_1.o` a partir de `ppal_1.cpp`
2. Generar el objeto `opers.o` a partir de `opers.o`
3. Generar el ejecutable `ppal_1` a partir de los dos módulos objeto generados.

Ejercicio

Escribir un fichero llamado `makefile_ppal_1` para generar el ejecutable `ppal_1`, siguiendo el diagrama de dependencia mostrado en la figura 1.

Usad macros para especificar los directorios de trabajo.

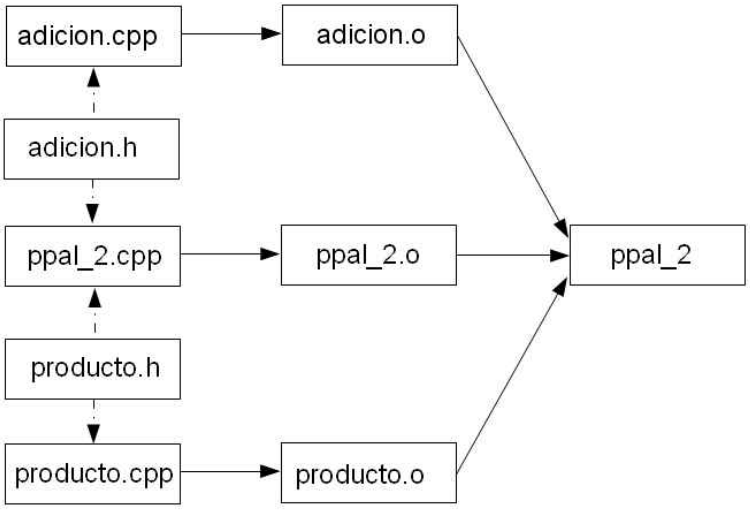

Figura 2: Diagrama de dependencias para construir `ppal_2`

Ejercicio

Continuaremos distribuyendo el código fuente en distintos ficheros, siguiendo el esquema indicado en la figura 2.

1. El fichero (`ppal_2.cpp`) contendrá la función `main()`
2. El código asociado a las funciones se organiza en cuatro ficheros organizados *en pares declaración-definición*:
   a) El primer par (`adicion.h` y `adicion.cpp`) contiene las funciones `suma()` y `resta()`
   b) El segundo (`producto.h` y `producto.cpp`) contiene las funciones `multiplica()` y `divide()`

Realizar las siguientes tareas:

1. Generar el objeto `ppal_2.o` a partir de `ppal_2.cpp`
2. Generar los objetos `adicion.o` y `producto.o`
3. Generar el ejecutable `ppal_2` a partir de los tres módulos objeto generados.

Ejercicio

Modificar la función `divide()` de `producto.cpp` de manera que procese adecuadamente la excepción que se genera cuando hay una división por cero.

1. Analizar qué módulos deben recompilarse para generar un nuevo ejecutable, actualizado con la modificación propuesta.
2. Volver a generar el ejecutable `ppal_2`

Ejercicio

Escribir un fichero llamado `makefile_ppal_2` para generar el ejecutable `ppal_2`, siguiendo el diagrama de dependencia mostrado en la figura 2.

Usad macros para especificar los directorios de trabajo.

2. Bibliotecas

En la práctica de la programación se crean conjuntos de funciones útiles para muy diferentes programas: funciones de depuración de entradas, funciones de presentación de datos, funciones de cálculo, etc. Si cualquiera de esas funciones quisiera usarse en un nuevo programa, la práctica de "Copiar y Pegar" el trozo de código correspondiente a la función deseada no resulta, a la larga, una buena solución, ya que,

1. El tamaño total del código fuente de los programas se incrementa innecesariamente y es redundante.
2. Si se hace necesaria una actualización del código de una función es preciso modificar **TODOS** los programas que usan esta función, lo que llevará a utilizar diferentes versiones de la misma función si el control no es muy estricto o a un esfuerzo considerable para mantener la coherencia del software.

En cualquier caso, esta práctica llevará irremediablemente en un medio/largo plazo a una situación insostenible. La solución obvia consiste en agrupar estas funciones usadas frecuentemente en módulos de biblioteca, llamados comúnmente **bibliotecas**.

*Una **biblioteca** contiene código objeto que puede ser enlazado con el código objeto de un módulo que usa una función de esa biblioteca*.

De esta forma tan solo existe una versión de cada función por lo que la actualización de una función implicará únicamente recompilar el módulo donde está esa función y los módulos que usan esa función, sin necesidad de modificar nada más (siempre que no se modifique la cabecera de la función, y como consecuencia, la llamada a ésta, claro está). Si además nuestros proyectos se matienen mediante ficheros makefile el esfuerzo de mantenimiento y recompilación se reduce drásticamente.
Esta **modularidad** redunda en un mantenimiento más eficiente del software y en una disminución del tamaño de los ficheros fuente, ya que tan sólo incluirán la llamada a las funciones de biblioteca, y no su definición.

Vulgarmente se emplea el término *librería* para referirse a una biblioteca, por la similitud con el original inglés *library*. En términos formales, la acepción correcta es **biblioteca**, porque es la traducción correcta de **library**, mientras que el término inglés para librería es *bookstore* o *book shop*. También es habitual referirse a ella con el término de origen anglosajón *toolkit* (conjunto, equipo, maletín, caja, estuche, juego (kit) de herramientas).

2.1. Tipos de bibliotecas

Bibliotecas estáticas

La dirección real, las referencias para saltos y otras llamadas a las funciones de las bibliotecas se almacenan en una *dirección relativa* o *simbólica*, que no puede resolverse hasta que todo el código es asignado a direcciones estáticas finales.

El enlazador resuelve todas las direcciones no resueltas convirtiéndolas en direcciones fijas, o relocalizables desde una base común. El enlace estático da como resultado un archivo ejecutable con todos los símbolos y módulos respectivos incluidos en dicho archivo. Este proceso se realiza antes de la ejecución del programa y debe repetirse cada vez que alguno de los módulos es recompilado.

La ventaja de este tipo de enlace es que hace que un programa no dependa de ninguna biblioteca (puesto que las enlazó al compilar), haciendo más fácil su distribución.

Bibliotecas dinámicas

Enlace dinámico significa que los módulos de una biblioteca son cargadas en un programa en tiempo de ejecución, en lugar de ser enlazadas en tiempo de compilación, y se mantienen como archivos independientes separados del fichero ejecutable del programa principal.

El enlazador realiza una mínima cantidad de trabajo en tiempo de compilación, registra qué modulos de la biblioteca necesita el programa y el índice de nombres de los módulos en la biblioteca.

Algunos sistemas operativos sólo pueden enlazar una biblioteca en tiempo de carga, antes de que el proceso comience su ejecución, otros son capaces de esperar hasta después de que el proceso haya empezado a ejecutarse y enlazar la biblioteca sólo cuando efectivamente se hace referencia a ella (es decir, en tiempo de ejecución). Esto último se denomina retraso de carga". En cualquier caso, esa biblioteca es una biblioteca enlazada dinámicamente.

2.2. Estructura de una biblioteca

Una biblioteca se estructura internamente como un **conjunto de módulos objeto**. Cada uno de estos módulos será el resultado de la compilación de un fichero de código fuente que puede contener una o varias funciones.

La extensión por defecto de los ficheros de biblioteca es `.a` y se acostumbra a que su nombre empiece por el prefijo `lib`. Así, por ejemplo, si hablamos de la biblioteca `ejemplo` el fichero asociado se llamará `libejemplo.a`

Veamos, con un ejemplo, cómo se estructura una biblioteca. La biblioteca `libejemplo.a` contiene 10 funciones. Los casos extremos en la construcción de esta biblioteca serían:

1. Está formada por *un único fichero objeto* (por ejemplo, `funcs.o`) que es el resultado de la compilación de un fichero fuente (`funcs.cpp`). Este caso se ilustra en la figura 3.

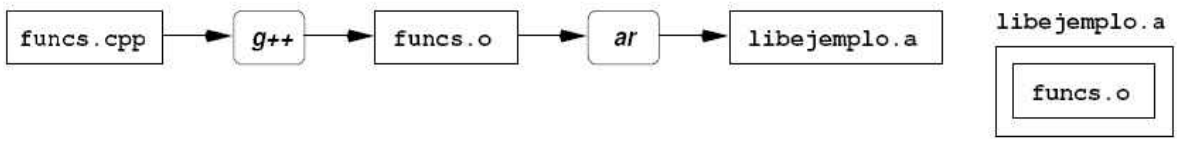

Figura 3: Construcción de una biblioteca a partir de un único módulo objeto (caso 1)

```
// funcs.cpp
// Contiene la definicion de 10 funciones
//

int funcion_1 (int a, int b)
{
   ...........
}

char *funcion_2 (char *s, char *t)
{
   ...........
}

...........

int funcion_10 (char *s, int x)
{
   ...........
}
```

2. Está formada por 10 ficheros objeto (por ejemplo, `fun01.o`, ..., `fun10.o`) resultado de la compilación de 10 ficheros fuente (por ejemplo, `fun01.cpp`, ..., `fun10.cpp`) que contienen, cada uno, la definición de una única función. Este caso se ilustra en las figuras 4 y 5.

fun01.cpp

```
// fun01.cpp
// Contiene la definicion de "funcion_1()"
//

int funcion_1 (int a, int b)
{
   ...........
}
```

fun10.cpp

```
// fun10.cpp
// Contiene la definicion de "funcion_10()"
//

int funcion_10 (char *s, int x)
{
   ...........
}
```

Figura 4: Varios módulos fuente (caso 2)

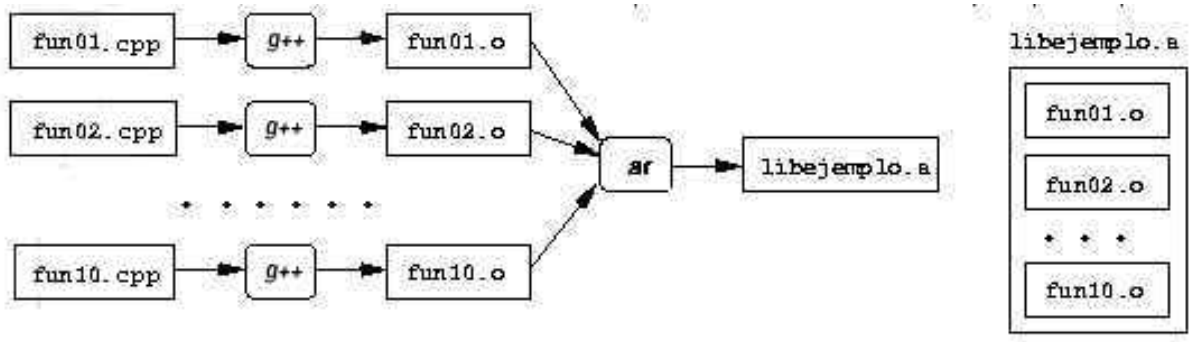

Figura 5: Construcción de una biblioteca a partir de varios módulos objeto (caso 2)

Para generar un fichero ejecutable que utiliza una función de una biblioteca, el enlazador **enlaza el módulo objeto que contiene la función** `main()` **con el módulo objeto (completo) de la biblioteca donde se encuentra la función utilizada**. De esta forma, *en el ejecutable sólo se incluirán los módulos objeto que contienen alguna función llamada por el programa*.

Por ejemplo, supongamos que `ppal.cpp` contiene la función `main()`. Esta función usa la función `funcion_2()` de la biblioteca `libejemplo.a`. Independientemente de la estructura de la biblioteca, `ppal.cpp` se escribirá de la siguiente forma:

```
#include "ejemplo.h" // prototipos de las funciones de "libejemplo.a"

int main (void)
{
   ......
   cad1 = funcion_2 (cad2, cad3);
   ......
}
```

Como regla general **cada biblioteca llevará asociado un fichero de cabecera** que contendrá los **prototipos** de las funciones que se ofrecen en la biblioteca (**funciones públicas**). Este fichero de cabecera actúa de *interface* entre las funciones de la biblioteca y los programas que la usan.

En nuestro ejemplo, independientemente de la estructura interna de la biblioteca, ésta ofrece 10 funciones cuyos prototipo se encuentran declarados en `ejemplo.h`.

Para la elección de la estructura óptima de la biblioteca hay que recordar que la parte de la biblioteca que se enlaza al código objeto de la función `main()` es el módulo objeto **completo** en el que se encuentra la función de biblioteca usada. En la figura 6 mostramos cómo se construye un ejecutable a partir de un módulo objeto (`ppal.o`) que contiene la función `main()` y una biblioteca (`libejemplo.a`).

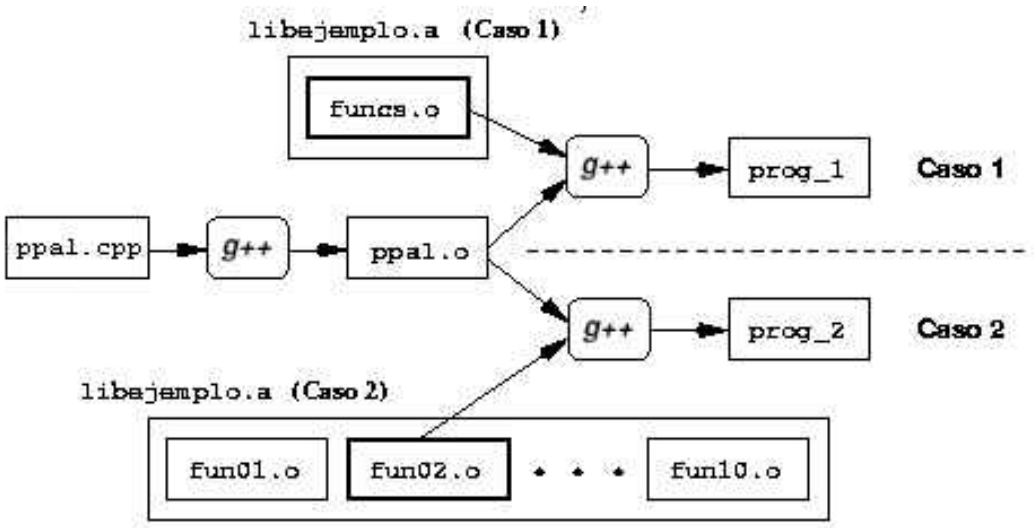

Figura 6: Construcción de un ejecutable que usa una función de biblioteca. Caso 1: biblioteca formada por un módulo objeto. Caso 2: biblioteca formada por 10 módulos objeto

Sobre esta figura, distinguimos dos situaciones diferentes dependiendo de cómo se construye la biblioteca. En ambos casos el fichero objeto que se enlazará con la biblioteca (con más precisión, con el módulo objeto adecuado de la biblioteca) se construye de la misma forma: el programa que usa una función de biblioteca no conoce (ni le importa) cómo se ha construído la biblioteca.

- **Caso 1**: El módulo objeto `ppal.o` se enlaza con el módulo objeto `funcs.o` (extraido de la biblioteca `libejemplo.a`) para generar el ejecutable `prog_1`. El módulo `funcs.o` contiene el código objeto de **todas** las funciones, por lo que se enlaza mucho código que no se usa.
- **Caso 2**: El módulo objeto `ppal.o` se enlaza con el módulo objeto `fun02.o` (extraido de la biblioteca `libejemplo.a`) para generar el ejecutable `prog_2`. El módulo `fun02.o` contiene **únicamente** el código objeto de la función que se usa, por lo que se enlaza el código estrictamente necesario.

En cualquier caso, como **usuario** de la biblioteca se actúa de la misma manera en ambas situaciones: enlaza el módulo objeto que conntiene la función `main()` con la biblioteca.
Como **diseñador** de la biblioteca sí debemos tener en cuenta la estructura que vamos a adoptar para ésta. La idea básica es conocida: si las bibliotecas están formadas por módulos objeto con muchas funciones cada una, los tamaños de los ejecutables serán muy grandes. En cambio, si las bibliotecas están formadas por módulos objeto con pocas funciones cada una, los tamaños de los ejecutables serán más pequeños[1].

3. El programa `ar`

El programa gestor de bibliotecas de GNU es `ar`. Con este programa es posible crear y modificar bibliotecas existentes: añadir nuevos módulos objeto, eliminar módulos objeto o reemplazarlos por otros más recientes. La sintaxis de la llamada a `ar` es:

`ar` [-]*operación* [*modificadores*] *biblioteca* [*módulos objeto*]

donde:

- *operación* indica la tarea que se desea realizar sobre la biblioteca. Éstas pueden ser:
  - `r` **Adición o reemplazo**. Reemplaza el módulo objeto de la biblioteca por la nueva versión. Si el módulo no se encuentra en la biblioteca, se añade a la misma. Si se emplea, además, el modificador `v`, `ar` imprimirá una línea por cada módulo añadido o reemplazado, especificando el nombre del módulo y las letras `a` o `r`, respectivamente.
  - `d` **Borrado**. Elimina un módulo de la biblioteca.
  - `x` **Extracción**. Crea el fichero objeto cuyo nombre se especifica y copia su contenido de la biblioteca. La biblioteca queda inalterada.
    Si no se especifica ningún nombre, se extraen todos los ficheros de la biblioteca.
  - `t` **Listado**. Proporciona una lista especificando los módulos que componen la biblioteca.
- *modificadores*: Se pueden añadir a las operaciones, modificando de alguna manera el comportamiento por defecto de la operación. Los más importantes (y casi siempre se utilizan) son:
  - `s` **Indexación**. Actualiza o crea (si no existía previamente) el índice de los módulos que componen la biblioteca. *Es necesario que la biblioteca tenga un índice para que el enlazador sepa cómo enlazarla*. Este modificador puede emplearse acompañando a una operación o por si solo.
  - `v` **Verbose**. Muestra información sobre la operación realizada.
- *biblioteca* es el nombre de la biblioteca a crear o modificar.
- *módulos objeto* es la lista de ficheros objeto que se van a añadir, eliminar, actualizar, etc. en la biblioteca.

Los siguientes ejercicios ayudarán a entender el funcionamiento de las distintas opciones de `ar`.

[1] Estas afirmaciones suponen que las funciones son equiparables en tamaño, obviamente

Ejercicio

Repetir los siguientes ejemplos:

1. Crear la biblioteca `libprueba.a` a partir del módulo objeto `opers.o`.

```
ar -rvs lib/libprueba.a obj/opers.o
```

2. Mostrar los módulos que la componen.

```
ar -tv lib/libprueba.a
```

Ejercicio

1. Añadir los módulos `adicion.o` y `producto.o` a la biblioteca `libprueba.a`.

```
ar -rvs lib/libprueba.a obj/adicion.o obj/producto.o
```

2. Mostrar los módulos que la componen.

```
ar -tv lib/libprueba.a
```

Ejercicio

1. Extraer el módulo `adicion.o` de la biblioteca `libprueba.a`.

```
ar -xvs lib/libprueba.a adicion.o
```

2. Mostrar los módulos que la componen.

```
ar -tv lib/libprueba.a
```

3. Mostrar el directorio actual.

Ejercicio

1. Borrar el módulo `producto.o` de la biblioteca `libprueba.a`.

```
ar -dvs lib/libprueba.a producto.o
```

2. Mostrar los módulos que la componen.

```
ar -tv lib/libprueba.a
```

3. Mostrar el directorio actual.

4. `g++`, `make` y `ar` trabajando conjuntamente

Como hemos señalado, `ar` es un programa general que empaqueta/desempaqueta ficheros en/desde archivos, de manera similar a como hacen otros programas como *zip*, *rar*, *tar*, etc. (una lista amplia puede encontrarse en `http://es.wikipedia.org/wiki/Anexo:Archivadores_de_ficheros`).

Nuestro interés es aprender cómo puede emplearse una biblioteca para la generación de un ejecutable, y cómo escribir las dependencias en ficheros makefile para poder automatizar todo el proceso de creación/actualización con la orden `make`.

4.1. Creación de la biblioteca

Emplearemos una regla para la construcción de la biblioteca.

1. El *objetivo* será la biblioteca a construir.
2. La *lista de dependencias* estará formada por los módulos objeto que constituirán la biblioteca.

Para los dos casos estudiados en la sección 2.2 escribiríamos:

1. Caso 1.

```
lib/libejemplo.a : obj/funcs.o
    ar -rvs /libejemplo.a obj/funcs.o
```

2. Caso 2.

```
lib/libejemplo.a : obj/fun01.o obj/fun02.o obj/fun03.o obj/fun04.o\
                   obj/fun05.o obj/fun06.o obj/fun07.o obj/fun08.o\
                   obj/fun09.o obj/fun10.o
    ar -rvs lib/libejemplo.a obj/fun01.o obj/fun02.o obj/fun03.o\
                   obj/fun04.o obj/fun05.o obj/fun06.o obj/fun07.o\
                   obj/fun08.o obj/fun09.o obj/fun10.o
```

(La barra simple invertida (`\`) es muy útil para dividir en varias líneas órdenes muy largas ya que permite continuar escribiendo en una nueva línea la misma orden)

Evidentemente, resulta aconsejable escribir de manera más compacta y generalizable esta regla:

```
MODS_EJEMPLO = obj/fun01.o obj/fun02.o obj/fun03.o obj/fun04.o\
               obj/fun05.o obj/fun06.o obj/fun07.o obj/fun08.o\
               obj/fun09.o obj/fun10.o
......
lib/libejemplo.a : $(MODS_EJEMPLO)
    ar -rvs lib/libejemplo.a $(MODS_EJEMPLO)
```

4.2. Enlazar con una biblioteca

El enlace lo realiza `g++`, llamando al enlazador `ld`. Extenderemos la regla que genera el ejecutable:

1. El *objetivo* es el mismo: generar el ejecutable.
2. La *lista de dependencias* incluirá ahora a la biblioteca que se va a enlazar.
3. La *orden* debe especificar:
   a) El directorio dónde buscar la biblioteca (opción `-L`)
   Recordemos que la opción `-L`*path* indica al enlazador el directorio donde se encuentran los ficheros de biblioteca. Se puede utilizar la opción `-L` varias veces para especificar distintos directorios de biblioteca.
   b) El nombre (resumido) de la biblioteca (opción `-l`).
   Los ficheros de biblioteca se proporcionan a `g++` de manera resumida, escribiendo `-l`*nombre* para referirnos al fichero de biblioteca `lib`*nombre*`.a` El enlazador busca en los directorios de bibliotecas (entre los que están los especificados con `-L`) un fichero de biblioteca llamado `lib`*nombre*`.a` y lo usa para enlazarlo.

Para los dos casos estudiados en la sección 2.2 escribiríamos:

1. Caso 1.

```
bin/prog_1 : obj/ppal.o lib/libejemplo.a
    g++ -o bin/prog_1 obj/ppal.o -L./lib -lejemplo
```

2. Caso 2.

```
bin/prog_2 : obj/ppal.o lib/libejemplo.a
    g++ -o bin/prog_2 obj/ppal.o -L./lib -lejemplo
```

5. Ejercicios

Para poder realizar los ejercicios propuestos es necesario disponer de los ficheros:

- `ppal_1.cpp`, `ppal_2.cpp`, `opers.cpp`, `adicion.cpp` y `producto.cpp`
  Usaremos un nuevo fichero fuente, `ppal.cpp`, que será una copia de `ppal_1.cpp` Este fichero se usará en todos los ejercicios, y únicamente cambiará(n) la(s) línea(s) `#include`
- `opers.h`, `adicion.h` y `producto.h`
- `makefile_ppal_1` y `makefile_ppal_2`

En todos los ejercicios debe poder diferenciar claramente los dos actores que pueden intervenir:

1. El creador de la biblioteca
2. El usuario de la bibilioteca

aunque sea la misma persona (en este caso, usted) quien realice las dos tareas. Esta distinción es fundamental cuando se maneje los ficheros de cabecera: puede llegar a manejar dos ficheros exactamente iguales aunque con nombre diferentes. Uno de ellos será empleado por el creador de la biblioteca y una vez construida la biblioteca, proporcionará otro fichero de cabecera que sirva de interface de la biblioteca a los usuarios de la misma.

Ejercicio

Utilizar como base `makefile_ppal_1` y construir el fichero makefile llamado `makefile_1lib_1mod` que implementa el diagrama de la figura 7

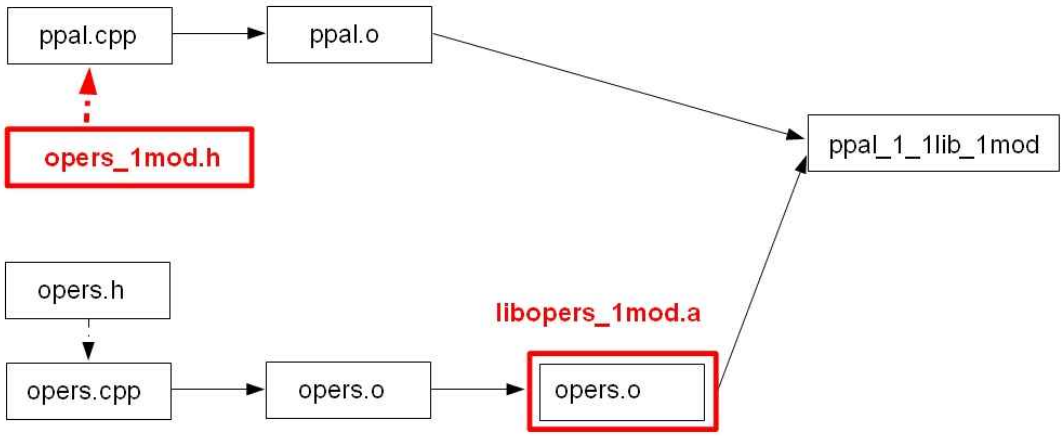

Figura 7: Construcción de un ejecutable que usa funciones de biblioteca (1). Una biblioteca formada por un módulo objeto que implementa cuatro funciones.

En este caso tenemos una biblioteca formada por un único módulo objeto. Todas las funciones están definidas en ese módulo objeto. Observe que el ejecutable será exactamente igual que el que llamamos anteriormente `ppal_1` (ver figura 3 de la práctica 2). Explique por qué.

El fichero de cabecera `opers_1mod.h` asociado a la biblioteca `libopers_1mod.a` podría tener el mismo contenido que el que se emplea para construir la biblioteca.

Ejercicio

Utilizar como base `makefile_ppal_2` y construir el fichero makefile llamado `makefile_1lib_2mod` que implementa el diagrama de la figura 8

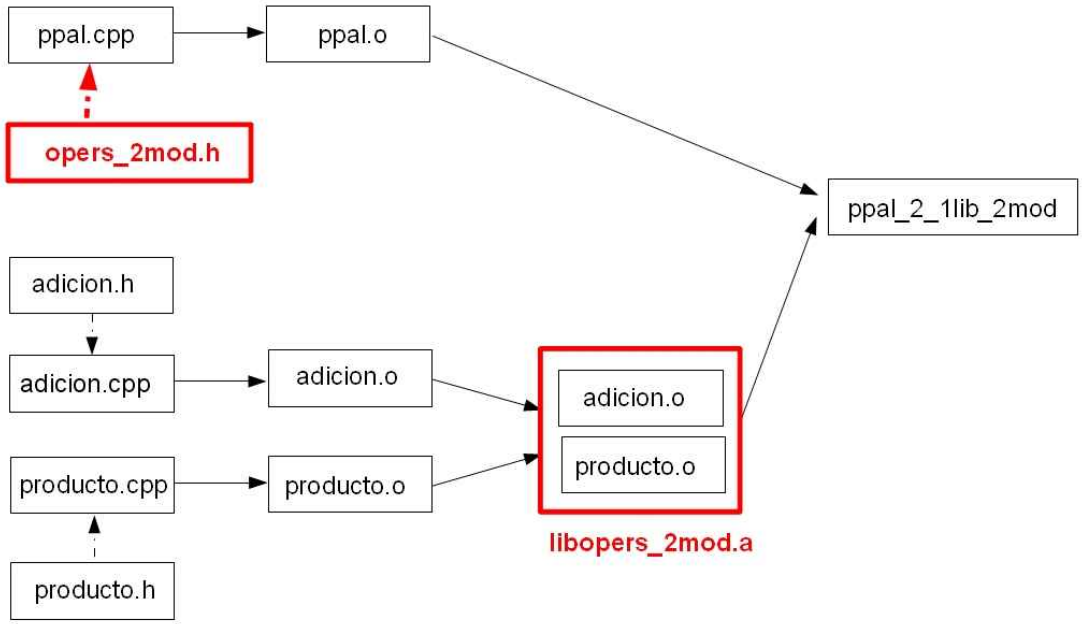

Figura 8: Construcción de un ejecutable que usa funciones de biblioteca (2). Una biblioteca formada por dos módulos objeto que implementan dos funciones cada uno de ellos.

En este caso tenemos una biblioteca formada por dos módulos objeto. Cada uno define dos funciones. Observe que el ejecutable será exactamente igual que el que llamamos anteriormente `ppal_2` (ver figura 4 de la práctica 2). Explique por qué.

El fichero de cabecera `opers_2mod.h` asociado a la biblioteca `libopers_2mod.a` podría tener el mismo contenido que `opers_1mod.h` (ejercicio anterior) si se pretende ofrecer la misma funcionalidad.

Ejercicio

Construir el fichero makefile llamado `makefile_1lib_4mod`. El fichero makefile implementa el diagrama de dependencias mostrado en la figura 9 En este caso tenemos una biblioteca formada por cuatro módulos objeto, y cada uno define una única función.

Como trabajo previo deberá crear los ficheros fuente `suma.cpp`, `resta.cpp`, `multiplica.cpp` y `divide.cpp`. Observe que no se han considerado ficheros de cabecera para cada uno de éstos ¿por qué?

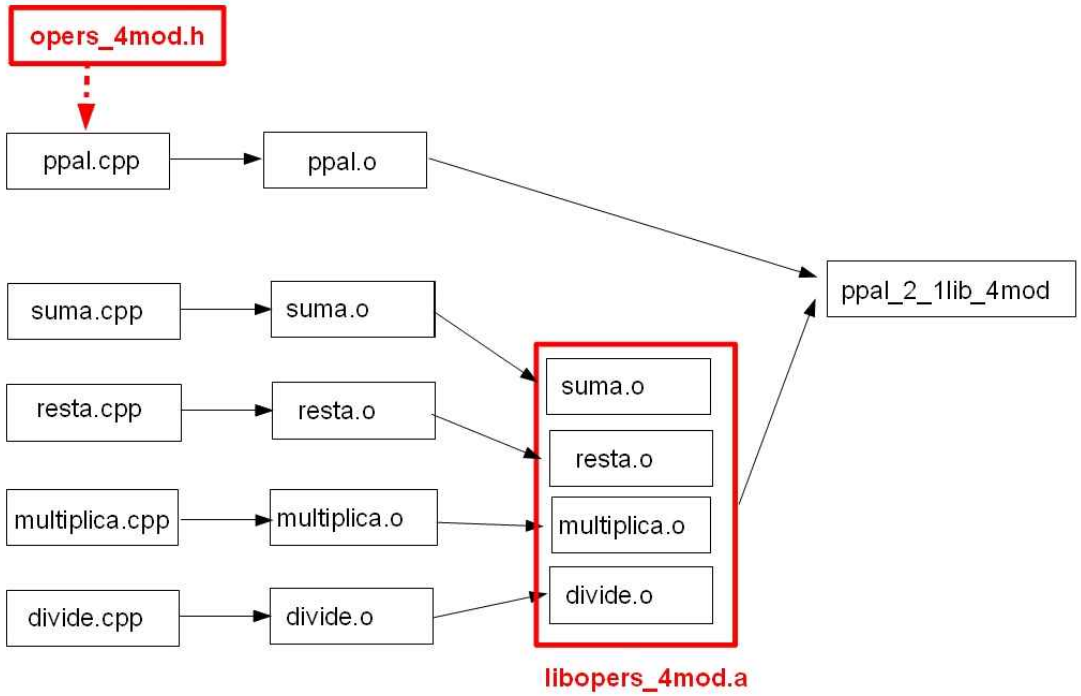

Figura 9: Construcción de un ejecutable que usa funciones de biblioteca (3). Una biblioteca formada por cuatro módulos objeto, y cada uno define una única función.

El fichero de cabecera `opers_4mod.h` asociado a la biblioteca `libopers_4mod.a` podría tener el mismo contenido que `opers_2mod.h` y `opers_1mod.h` (ejercicios anteriores) si se pretende ofrecer la misma funcionalidad.

Ejercicio

Construir el fichero makefile llamado `makefile_2lib_2mod`. El fichero makefile implementa el diagrama de dependencias mostrado en la figura 10

Observad que el ejecutable resultante debe ser exactamente igual que el anterior ¿por qué?

En este caso tenemos dos bibliotecas (`libadic_2mod.a` y `libproducto_2mod.a`) con sus ficheros de cabecera asociados (`adic_2mod.h` y `producto_2mod.h`). Cada una de las bibliotecas está compuesta de dos módulos objeto, y cada uno define dos funciones.

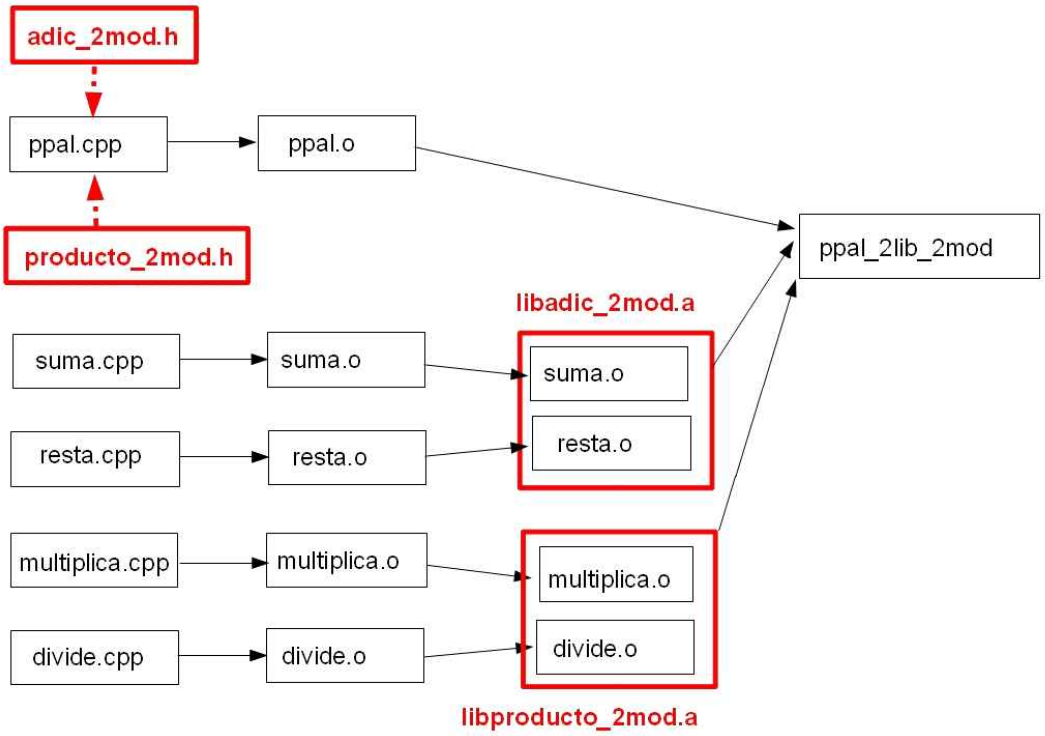

Figura 10: Construcción de un ejecutable que usa funciones de biblioteca (4). Dos bibliotecas formadas cada una por dos módulos objeto, y cada uno define dos funciones.

Ejercicio

1. Tomar nota de los tamaños de los ejecutables `ppal_1lib_1mod`, `ppal_1lib_2mod`, `ppal_1lib_4mod` y `ppal_2lib_2mod`.
2. Editar el fichero `ppal.cpp` y comentar la línea que llama a la función `suma()`.
3. Volver a construir los ejecutables `ppal_1lib_1mod`, `ppal_1lib_2mod`, `ppal_1lib_4mod` y `ppal_2lib_2mod`.
4. Anotar los tamaños de los ejecutables, compararlos con los anteriores y discutir.

Ejercicio

En la *Relación de problemas I (punteros)* propusimos escribir unas funciones para la gestión de cadenas clásicas de caracteres.
Los prototipos de las funciones son:

```
int longitud_cadena (const char * cadena);
bool es_palindromo (const char * cad);
int comparar_cadenas (const char * cad1, const char * cad2);
char * copiar_cadena (char * cad1, const char * cad2);
char * encadenar_cadena (char * cad1, const char * cad2);
```

1. Encapsular estas funciones en una biblioteca llamada `libmiscadenas.a` y escribir un módulo llamado `demo_cadenas.cpp` que contenga únicamente una función `main()` y que use las cuatro funciones.

   No olvide escribir el fichero de cabecera asociado a la biblioteca, y llamarle `miscadenas.h`

2. Escribir un fichero *makefile* para generar la biblioteca y el ejecutable.

3. Añadir a la biblioteca la función propuesta para extraer una subcadena, con prototipo

   ```
   char * subcadena (char * cad1, const char * cad2, int p, int l);
   ```

4. Editar `demo_cadenas.cpp` para que use la nueva función.

5. Volver a generar el ejecutable.

6. ¿Qué puede decirse del fichero de cabecera asociado a la biblioteca?