UNIVERSIDAD DE GRANADA

DECSAI

Sesión de Prácticas

Metodología de la Programación

Grado en Ingeniería Informática

David Pelta

Correo: dpelta@ugr.es

- Lunes de 09:00 a 12:00
- Martes de 09:00 a 12:00

ETSIIT, 4ta Planta, Despacho 21.

¿ Qué haremos ?

- Prueba frente al ordenador (mediados del cuatrimestre) (10%)
- Realización de un proyecto informático (final del cuatrimestre) (20%)
- Evaluación continua: asistencia + entrega de ejercicios a lo largo del cuatrimestre

  (10%)

Primer Paso

- Crear una estructura de directorios
- “mkdir”, “ls”, “pwd”

mp
- bin
- doc
- include
- lib
- obj
- src

- Con tu editor favorito (gedit, kwrite), escribe un programa del tipo “Hola Mundo” y guardalo en la carpeta *src*

```
#include <iostream>
using namespace std;

int main(){
 cout << "\n Hola Mundo \n";
}
```

Compilación

```
g++ -o nombre-ejecutable nombre-fuente
```

- Compilar y ejecutar nuestro ejemplo

```
g++ -o bin/holaMundo src/holaMundo.cpp

./bin/holaMundo
```

- Compilar en dos pasos
- Introduce errores en *holaMundo.cpp* y observa la salida del compilador

Pasos en la Compilación

El preprocesador (Está integrado en el compilador: g++ -E)

- Recibe el código fuente (.h y .cpp)
- Devuelve el código fuente preprocesado:
  - Elimina los comentarios
  - Interpreta y procesa las directivas del preprocesador

El compilador

- Recibe el código fuente preprocesado
- Análisis léxico, sintáctico, semántico, Optimización
- Devuelve el código objeto (casi código máquina)

El "Linker"

- Recibe código objeto y bibliotecas
- Devuelve el ejecutable (código máquina)
- Alguno de los ficheros objeto debe tener una función main()

Modularización

Problema grande -> Divide y Vencerás (módulos)

Modularizar permite:

- Trabajo en equipo
- Facilita depuración (aisla errores)
- Mejorar legibilidad y modificabilidad
- Elimina redundancias y facilita reusabilidad de código

Conceptos importantes:

- Abstracción (funcional, de datos)
- Encapsulamiento / Ocultamiento de información
- Acoplamiento
- Diseño descendente (top-down)
- Compilación separada

Concepto de Módulo

Por módulo, podemos entender

- Una función.
- Un fichero (que contiene varias funciones y datos).
- Un TDA (Tipo de Dato Abstracto).
- Una biblioteca (conjunto de ficheros y/o funciones y/o datos).
- Un namespace (agrupamiento lógico de funciones y datos).

Módulo

Cuando se construyen programas grandes, las funciones y estructuras de datos se dividen y agrupan, según su uso, en múltiples ficheros y bibliotecas.

Cada módulo suele tener:

- Interfaz de acceso al módulo (**parte pública**)
- Implementación del módulo (**parte privada**).

Tener varios ficheros permite

- La compilación separada
- Facilita la construcción y el mantenimiento de los programas
- Facilita el trabajo en equipo

Módulo

Interfaz (Parte pública)

- Debe ser conocida por cualquiera que desee usar el módulo.
- Se escribe en los ficheros de cabecera (.h)
- **Contenido:** #define públicos, Prototipos de funciones, Declaración de tipos públicos (struct, class, typedef, enum, …)

Implementación (Parte privada)

- Sólo conocida por el programador o diseñador del módulo.
- Se escribe en los ficheros de implementación (.cpp)
- **Contenido:** la implementación de lo que se ha declarado en la parte pública e implementaciones privadas.

La parte privada se transformará en un fichero objeto, o bien una biblioteca.

Ejemplo de Modularización

demo1.cpp

```cpp
#include <iostream>

double suma (double a, double b){
  return a + b;
}

double resta (double a, double b){
  return a - b;
}

double multiplica (double a, double b){
  return a * b;
}

double divide (double a, double b){
  return a / b;
}

using namespace std;
int main (int argc, char *argv[]){

  double a, b;
  cout << "Introduce el primer valor: ";
  cin >> a;
  cout << "Introduce el segundo valor: ";
  cin >> b;
  cout << "suma(" << a << ", " << b << ") = " << suma(a,b) << endl;
  cout << "resta(" << a << ", " << b << ") = " << resta(a,b) << endl;
  cout << "multiplica(" << a << ", " << b << ") = " << multiplica(a,b) << endl;
  cout << "divide(" << a << ", " << b << ") = " << divide(a,b) << endl;

  return 0;
}
```

demo2.cpp

```cpp
#include <iostream>

double suma (double a, double b);
double resta (double a, double b);
double multiplica (double a, double b);
double divide (double a, double b);

using namespace std;
int main (int argc, char *argv[]){

  double a, b;
  cout << "Introduce el primer valor: ";
  cin >> a;
  cout << "Introduce el segundo valor: ";
  cin >> b;
  cout << "suma(" << a << ", " << b << ") = " << suma(a,b) << endl;
  cout << "resta(" << a << ", " << b << ") = " << resta(a,b) << endl;
  cout << "multiplica(" << a << ", " << b << ") = " << multiplica(a,b) << endl;
  cout << "divide(" << a << ", " << b << ") = " << divide(a,b) << endl;

  return 0;
}

double suma (double a, double b){
  return a + b;
}

double resta (double a, double b){
  return a - b;
}

double multiplica (double a, double b){
  return a * b;
}

double divide (double a, double b){
  return a / b;
}
```

Ejemplo

1. Construiremos una version modular de *demo1.cpp*
2. Copiar *demo1.cpp* a *demo2.cpp* (en *src*)
3. Crear un fichero *oper2.h* (en *include*) que contendrá la declaración de las funciones (la interfaz del módulo)
4. Crear un nuevo fichero *oper2.cpp* (en *src*) que contenga la implementación de las funciones suma, resta, multiplicación y división (la implementación del módulo)
5. En *demo2.cpp* agregar una línea `#include "oper2.h"`

include/oper2.h

```
#ifndef OPER2_H
#define OPER2_H

double suma (double a, double b);
double resta (double a, double b);
double multiplica (double a, double b);
double divide (double a, double b);
#endif
```

src/oper2.cpp

```
#include <oper2.h>

double suma (double a, double b){
  return a + b;
}

double resta (double a, double b){
  return a - b;
}

double multiplica (double a, double b){
  return a * b;
}

double divide (double a, double b){
  return a / b;
}
```

src/demo2.cpp

```
#include <iostream>
#include "oper2.h"

using namespace std;
int main (int argc, char *argv[]){

  double a, b;
  cout << "Introduce el primer valor: ";
  cin >> a;
  cout << "Introduce el segundo valor: ";
  cin >> b;
  cout << "suma(" << a << ", " << b << ") = "
       << suma(a,b) << endl;
  cout << "resta(" << a << ", " << b << ") = "
       << resta(a,b) << endl;
  cout << "multiplica(" << a << ", " << b << ") = "
       << multiplica(a,b) << endl;
  cout << "divide(" << a << ", " << b << ") = "
       << divide(a,b) << endl;

  return 0;
}
```

Observar dependencias!

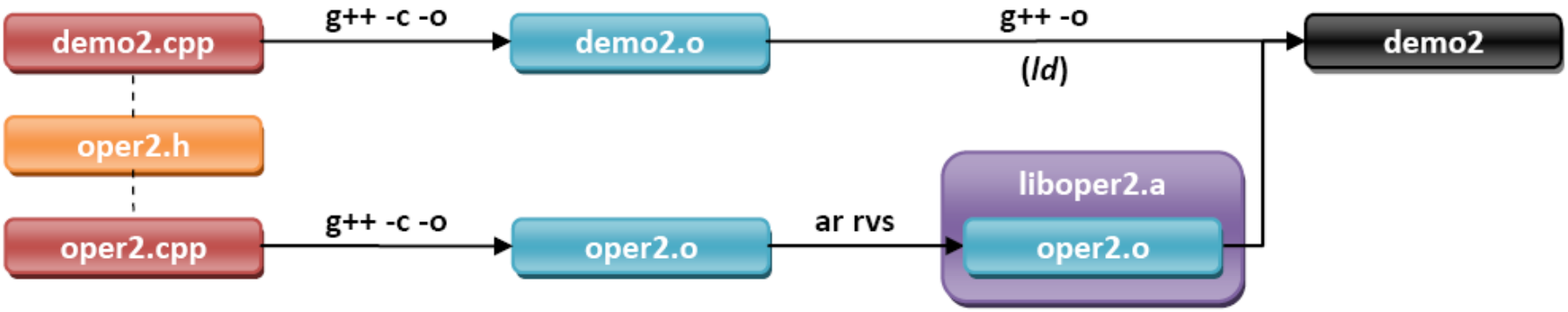
Compilación

- Se necesita el codigo objeto de cada fichero fuente.
- Opción –c del compilador
- Hacer `g++ -c -o obj/demo2.o src/demo2.cpp`
- Observar el error
- Hacer

```
g++ -c -o obj/demo2.o src/demo2.cpp -I./include
g++ -c -o obj/oper2.o src/oper2.cpp -I./include
g++ -o bin/demo2 obj/demo2.o obj/oper2.o
```

Construcción de una biblioteca

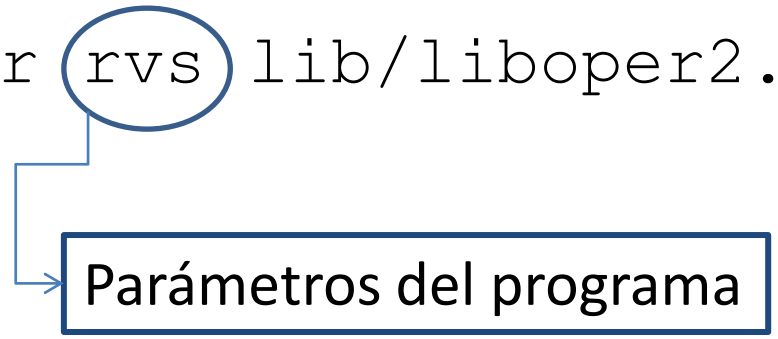
- Una biblioteca es un fichero que agrupa un conjunto de ficheros objeto (.o)
- En el ejemplo anterior el módulo se transformaba en código objeto que luego se enlazaba con el programa principal.
- Veamos como incluir el módulo en una biblioteca.
- El esquema es:



Construcción de una biblioteca

- La extensión por defecto de los ficheros de biblioteca es `.a`, y suelen comenzar con la palabra `lib`.
- Nuestra biblioteca
  - se llamará `liboper2.a`
  - la guardaremos en el directorio *lib*.
  - contendrá un solo fichero objeto `oper2.o`
- Las bibliotecas se crean con el programa *ar*

```
ar rvs lib/liboper2.a obj/oper2.o
```



Enlazando la biblioteca

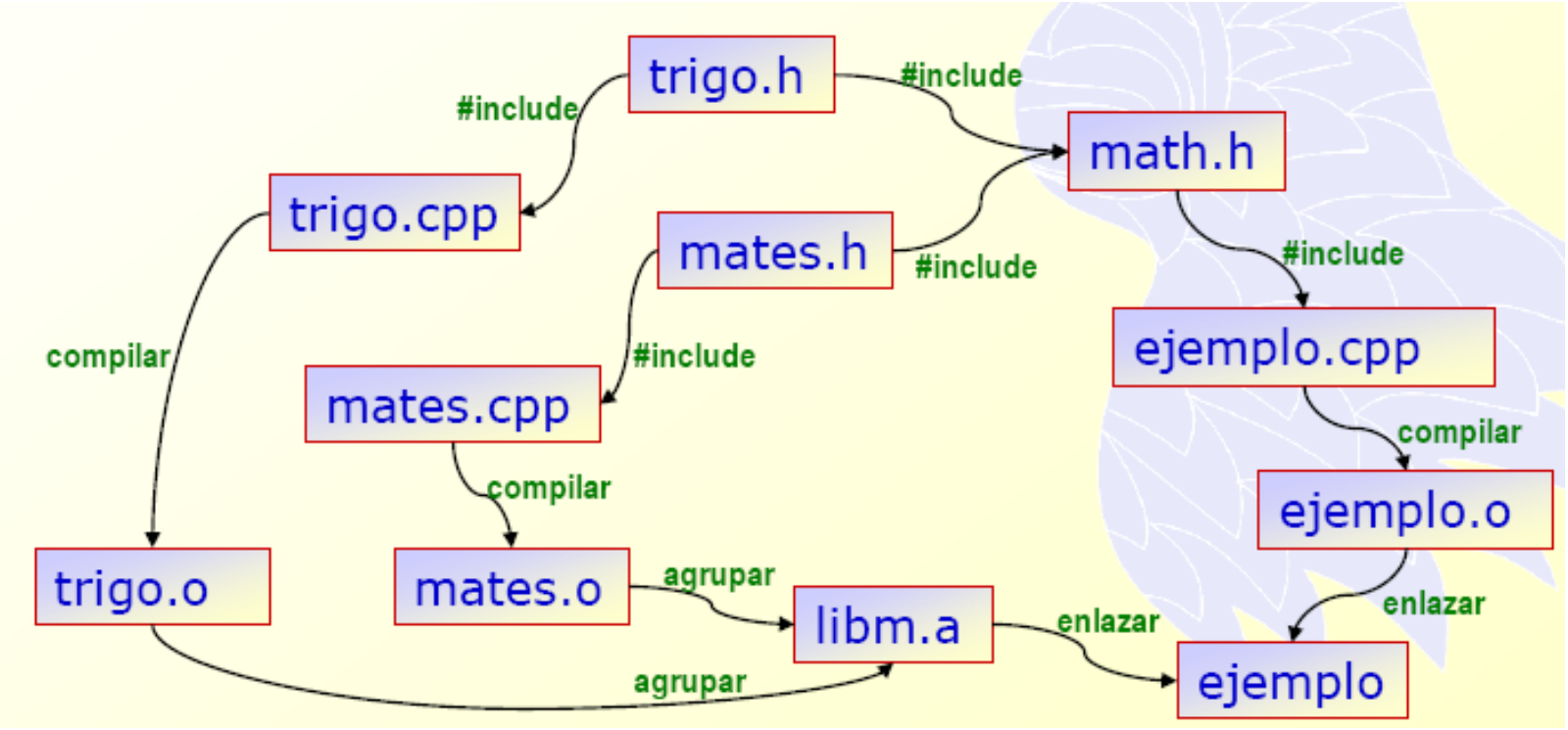
- Para construir el ejecutable con la biblioteca usamos la opción `-l` ('ele' minúscula, de library) de g++, seguida del nombre de la biblioteca quitando el lib del principio y el `.a` del final. Así en nuestro caso será `-loper2`.
- Usar la opción -L para especificar dónde hay que buscar la biblioteca, que para nuestro ejemplo será el directorio lib.

```
g++ -o bin/demo2 obj/demo2.o -L./lib -loper2
```

Cada vez que cambia un fichero fuente, deben recompilarse todos los ficheros que lo utilicen.

Un ejemplo más complejo

Supongamos que disponemos de la siguiente estructura de dependencias entre ficheros.

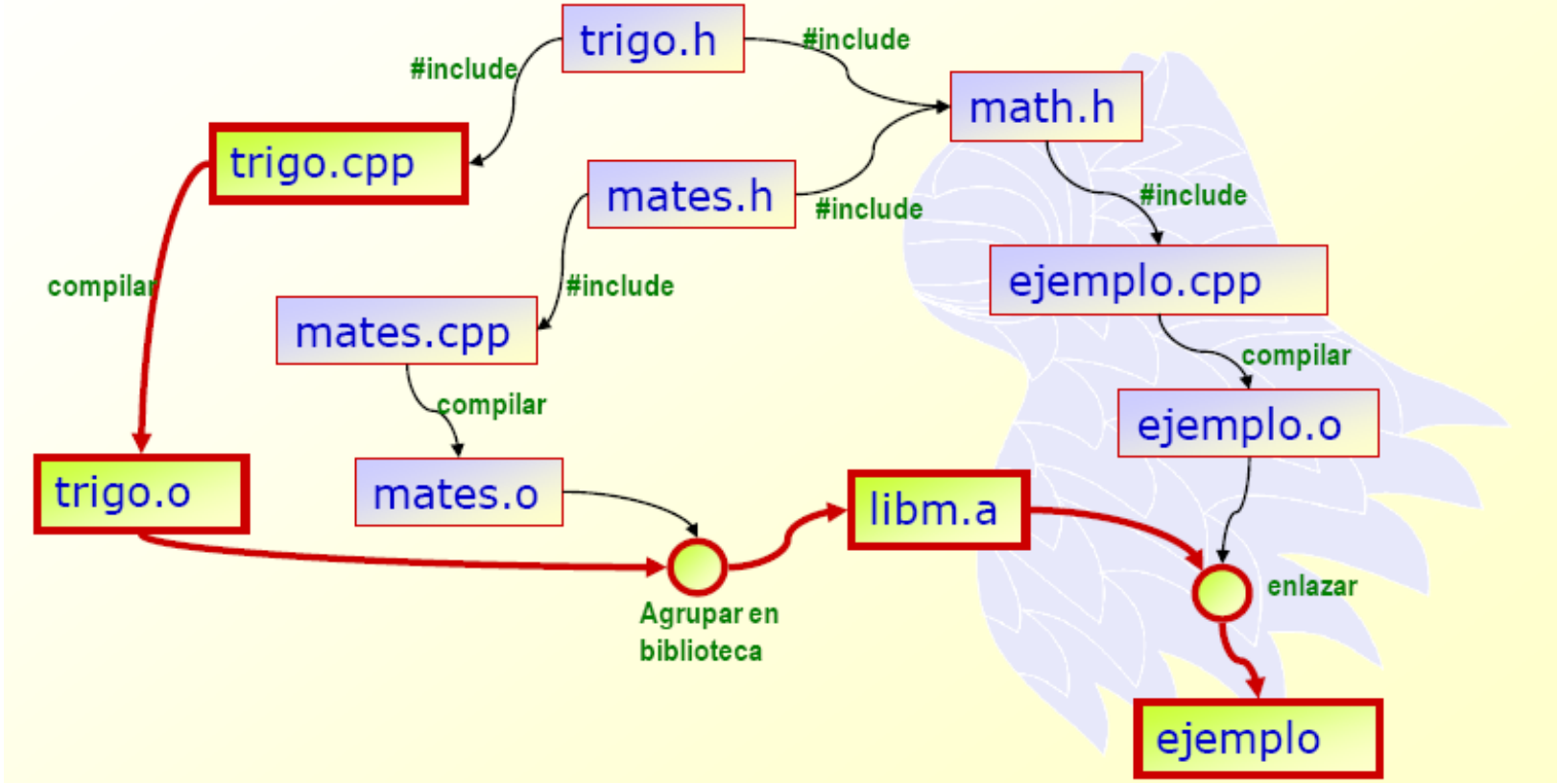


Ejemplo desarrollado por Javier Martinez Baena.

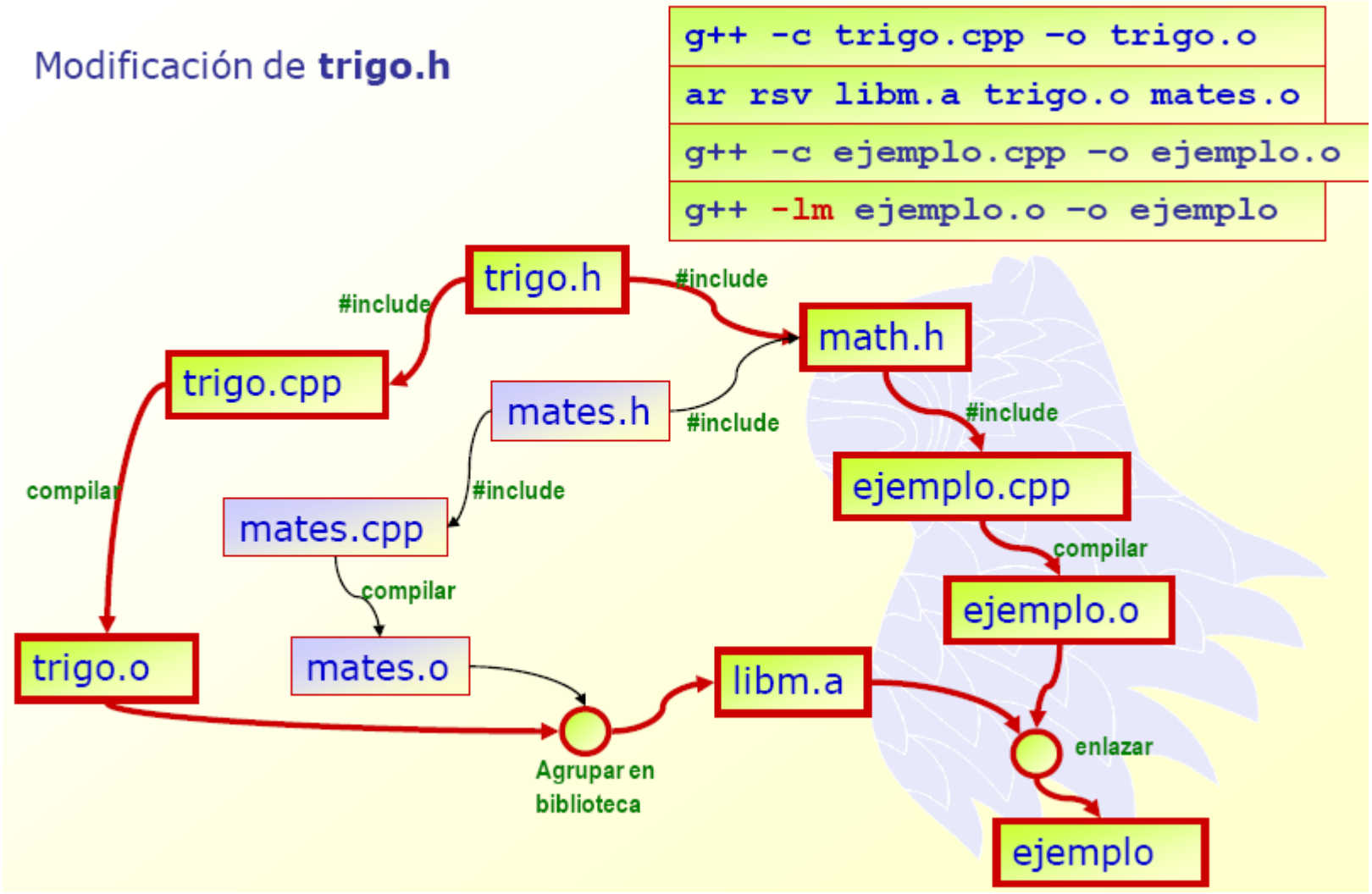
Primer Cambio

Modificación de **trigo.cpp**

```
g++ -c trigo.cpp -o trigo.o
ar rsv libm.a trigo.o mates.o
g++ -lm ejemplo.o -o ejemplo
```



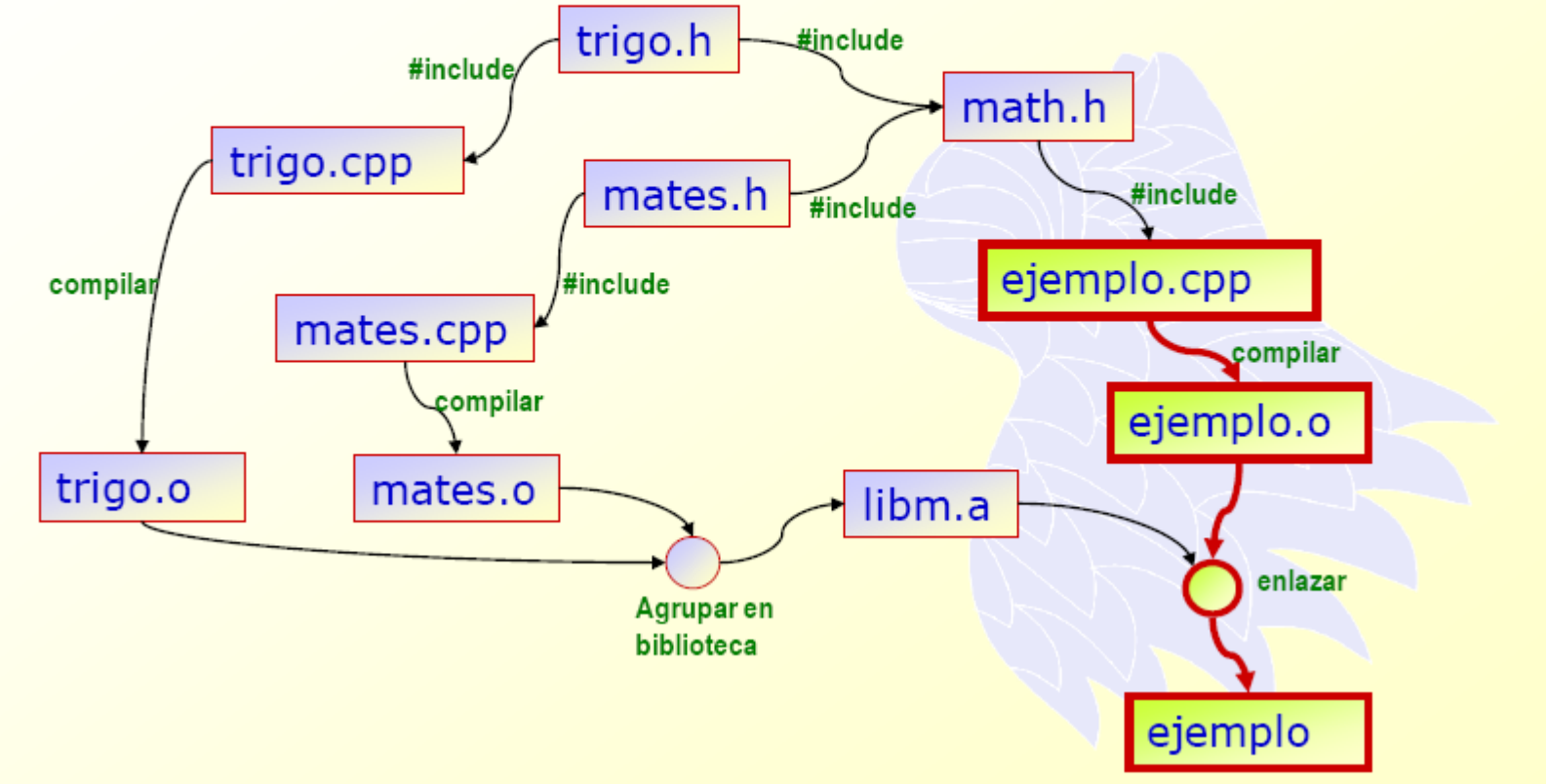
Segundo Cambio



Tercer Cambio

Modificación de **ejemplo.cpp**

```
g++ -c ejemplo.cpp -o ejemplo.o
g++ -lm ejemplo.o -o ejemplo
```



Automatización de la Compilación

- Las reglas de compilación y las dependencias entre ficheros se pueden detallar en un fichero *makefile*
- La utilidad *make* lee este fichero y realiza las tareas indispensables para reconstruir la aplicación.
- Dentro de un *makefile* es posible incluir una variedad muy amplia de comandos, reglas, variables, etc.
- En la guía práctica de compilación (disponible en la plataforma docente) se pueden ver las más importantes.

Ejemplo

Si observamos el gráfico de dependencias de derecha a izquierda, las reglas surgen naturalmente

destino simbólico

demo2, requiere demo2.o y oper2.o

Una vez que los tengo, ejecutar este comando

```
1  all : bin/demo2
2
3  bin/demo2 : obj/demo2.o obj/oper2.o
4  »        g++ -o bin/demo2 obj/demo2.o obj/oper2.o
5
6  obj/demo2.o : src/demo2.cpp include/oper2.h
7  »        g++ -c -I./include -o obj/demo2.o src/demo2.cpp
8
9  obj/oper2.o : src/oper2.cpp include/oper2.h
10 »        g++ -c -I./include -o obj/oper2.o src/oper2.cpp
11
```

Ejecución

- El fichero *makefile* lo guardamos en la carpeta *mp.*
- Luego ejecutamos *make* y automaticamente intentará construir el destino *all*
- También podemos indicar `make all`
- No es necesario que el fichero se llame *makefile.*
- Si utilizamos otro nombre, por ej. *misReglas* entonces haremos `make -f misReglas`

Versión con biblioteca

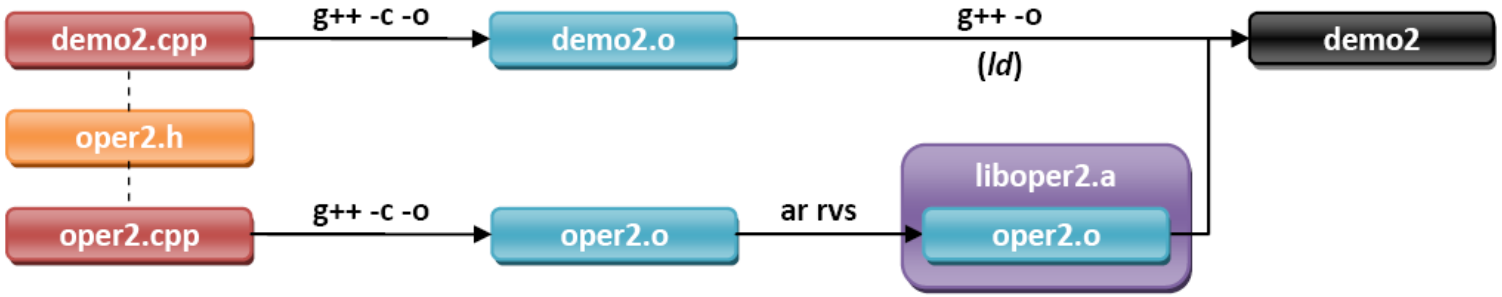
```
#fichero miMake
all : bin/demo2
bin/demo2 : obj/demo2.o lib/liboper2.a
     g++ -o bin/demo2 obj/demo2.o -L./lib -loper2

obj/demo2.o : src/demo2.cpp include/oper2.h
     g++ -c -I./include -o obj/demo2.o src/demo2.cpp

obj/oper2.o : src/oper2.cpp include/oper2.h
     g++ -c -I./include -o obj/oper2.o src/oper2.cpp

lib/liboper2.a: obj/oper2.o
   @echo construyendo libreria
   ar rvs lib/liboper2.a obj/oper2.o
clean:
   rm obj/*.o
```

```
make -f miMake
make -f miMake clean
```



Mas posibilidades

```
#fichero miMake2
INCLUDE = ./include
LIB = ./lib
OBJ = ./obj
SRC = ./src

all : $(BIN)/demo2

$(BIN)/demo2 : $(OBJ)/demo2.o $(LIB)/liboper2.a
     g++ -o $(BIN)/demo2 $(OBJ)/demo2.o -L$(LIB) -loper2

$(OBJ)/demo2.o : src/demo2.cpp include/oper2.h
     g++ -c -I$(INCLUDE) -o $(OBJ)/demo2.o $(SRC)/demo2.cpp

$(OBJ)/oper2.o : src/oper2.cpp include/oper2.h
     g++ -c -I$(INCLUDE) -o $(OBJ)/oper2.o $(SRC)/oper2.cpp

$(LIB)/liboper2.a: $(OBJ)/oper2.o
   @echo construyendo libreria
   ar rvs $(LIB)/liboper2.a $(OBJ)/oper2.o

clean :
   rm $(OBJ)/*.o
```

Macros para los nombres de directorios