Cómo hacer una librería

Tipos de archivos

`*.c`	Código fuente
`*.h`	Header files. Contiene declaraciones de funciones, macros, tipos, variables, etc.
`*.o`	Object files, archivos compilados pero no vinculados en un ejecutable
`*.a`	Librerías con archivos `.o`
`*`	Los ejecutables en Unix no tienen extensión (en Windows `.exe`)
`a.out`	Assembler Output. Nombre del ejecutable por defecto.

Librería estática

Vamos a hacer una librería estática, distinta de una shared library.
Cuando compilamos nuestros archivos `.c` se compilan a object files con extensión `.o`. Pero aunque se hagan referencia entre ellos hace falta usar un **linker** para combinar y conectar todos esos archivos entre sí.
Se puede compilar sin linking con la flag `-c`.

Una librería es una conjunto de archivos `.o` compilados pero no vinculados.
Se archivan con `ar`.

Compilar convierte archivos `.c` en archivos `.o`.
Linking combina archivos `.o` y devuelve un ejecutable.
`ar` recolecta archivos `.o`, sin linking, devuelve un archivo `.a`.

Cuando compilemos un programa que use funciones de esa librería, entonces sí que habrá un proceso de linking y se copiará el código necesario, no la librería entera.

cc y ar

Primero compilamos los archivos `.c` en archivos `.o`, pero **sin linking** (por eso ponemos `-c`):

```
cc -c -Wall -Werror -Wextra func1.c func2.c func3.c
```

Luego creamos la librería estática con `ar`:

```
ar rcs mylib.a func1.o func2.o func3.o
```

Las flags `rcs` significan insertar o reemplazar (`r`), crear si es necesario (`c`) y añadir índice para linking más rápido (`s`).
Los dos comandos anteriores deben hacerse en un `Makefile` (ver más adelante).

Libft

La Libft de 42 tiene los siguientes requisitos:
Compilar con `cc` y las flags `-Wall -Werror -Wextra`.
`Makefile` con variable `NAME` y reglas `all`, `clean`, `fclean`, `re`.
Los bonus se incluyen en la regla `bonus`.

Header files

Para hacer una librería haremos un archivo con extensión `.h`.
En el archivo se indican los prototipos de todas las funciones.

libft.h

```
#ifndef LIBFT_H
# define LIBFT_H
# include <stdlib.h>
# include <stdio.h>

int     ft_isalpha(int c);
int     ft_isdigit(int c);
int     ft_isalnum(int c);
// etc.

#endif
```

Header guards

Los header o include guards son condiciones que se expresan de esta manera:

```
#ifndef SOME_UNIQUE_NAME_HERE
# define SOME_UNIQUE_NAME_HERE

// bla bla bla

#endif
```

Esto sirve para comprobar si ya existe la definición. Si no existe (`ifndef` , if not defined), se define.

#define

La directriz de sustitución `#define XXX YYY` indica que `XXX` se debe sustituir por `YYY`.
Una **macro** es un fragmento de código que tiene un nombre. Por ejemplo:

```
#define PI 3.14159
```

Cada vez que el preprocesador encuentre `PI` lo reemplazará por `3.14159` antes de compilar.
Se pueden definir constantes, expresiones y expresiones con parámetros:

```
#define MACRO_NAME value

#define MACRO_NAME (expression within brackets)

#define MACRO_NAME(ARG1, ARG2, ...) (expression within brackets)
```

Cuando usamos `#define` y sólo el nombre (sin ningún valor) estamos creando una condición de compilación. Estamos creando una **flag** o un **marker**. Esto impide que el mismo archivo se compile varias veces.

#include

Para usar cualquier de esas funciones en otros archivos podemos importar la librería de dos maneras:

```
    # include "libft.h"
    # include <libft.h>
```

La notación entre comillas `""` indica que se buscará ese archivo en el directorio actual.
Si usamos `<>` el preprocesador buscará la librería en el directorio predefinido.

Makefile

GNU: GNU make
Christian Zapata Arias: Compile your programs faster with Makefile

Makefile es un archivo donde definimos reglas para compilar un programa con el comando `make`:

```
main: main.c stack.o
 cc -Wall -g -o main main.c stack.o

stack.o: stack.c stack.h
 cc -Wall -g -c stack.c

.PHONY: clean
clean:
 rm -rf *.o
```

Una regla tiene el formato `target: prerequisites`.
Normalmente target es el nombre del archivo que genera el programa o una acción como `clean`.
Los `prerequisites` son los archivos necesarios para crear el target. Si no los encuentra, buscará una regla para generarlos. Por ejemplo, en `main: main.c stack.o` se busca `stack.o` y, como no existe, se ejecuta la regla `stack.o`, cuyos prerrequisitos son `stack.c` y `stack.h`. Es decir, las reglas se pueden encadenar.

Para no repetir comandos podemos guardarlos en variables:

```
CC     = cc
CFLAGS = -Wall -g

main : main.c stack.o
 $(CC) $(CFLAGS) -o main main.c stack.o

stack.o : stack.c stack.h
 $(CC) $(CFLAGS) -c stack.c
```

Phony targets

Podemos crear comandos adicionales, pero para que no se confundan con nombres de archivos se prefija con `.PHONY`. Por ejemplo, un comando `clean` para borrar archivos:

```
.PHONY: clean
clean:
 rm -rf *.o
```

Bastará con ejecutar `make clean` para borrar los archivos con extensión `*.o`.

Si prefijamos un comando con `@` se ejecutará pero no se mostrará el comando en la consola.

all, clean, fclean, re

`NAME` Nombre del ejecutable final
`all` Default target, hace el build de todo, del ejecutable final, lo que se ejecuta al hacer `make`.
`clean` Borra archivos `.o` y otros archivos intermedios, pero preserva el ejecutable final.
`fclean` Full clean, borra todo incluso el ejecutable final.
`re` Reconstruye: borra todo (`fclean`) y hace un build con `all`.