

Compilación separada y ficheros makefile

Metodología de la Programación

Grado en Ingeniería Informática

Gabriel Navarro

(gnavarro@ugr.es, gnavarro@decsai.ugr.es)



**UNIVERSIDAD
DE GRANADA**

Curso 2017/2018

- Motivación
- División en varios ficheros
- El proceso de compilación
 - El preprocesador
 - El compilador
 - El enlazador
 - Sintaxis del compilador g++
- Construir bibliotecas. El programa ar
- El programa make. Ficheros makefile

Si tenemos un proyecto de **varios millones de líneas de código** en un sólo fichero, podemos tener algunos problemas:

- Para **encontrar componentes** del programa
- Para **corregir errores** en el código
- Para comprobar si algún módulo está correctamente implementado
- Para **modificar** el programa
- Para **reutilizar** algunas partes del programa en otro
- Para **trabajar en equipo** al implementarlo
- Para **ocultar la implementación** de algunas componentes
- Para **dar la documentación** a los usuarios

Una solución es **dividir el código** en varios archivos de código fuente y compilar cada uno por separado:

- Al estar más ordenado
 - Más fácil encontrar los errores
 - Más sencillo modificar el programa
- Al estar en varios archivos
 - Se puede ocultar la implementación de algún módulo al usuario
 - Es sencillo crear una documentación para el usuario
 - Es más fácil la reutilización del software
 - Se permite la posibilidad de trabajar en equipo ya que cada miembro se encarga de distintos archivos
 - Es más fácil la POO, cada objeto en un fichero distinto

División del código en ficheros

¿Cómo dividimos el código? Hasta ahora, un fichero `.cpp` listo para compilar, tiene el siguiente aspecto:

Inclusión de bibliotecas , por ejemplo <code>#include<iostream></code>
Espacio de nombres , normalmente <code>using namespace std;</code>
Constantes y variables globales
Tipo de datos definidos por el usuario , <code>struct Hora {...};</code>
Prototipos de las funciones
Función main
Definiciones de las funciones

División del código en ficheros

Extracción del código de las **funciones**

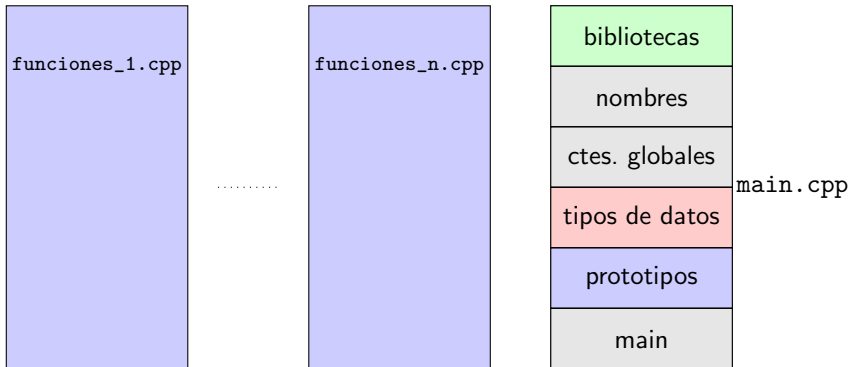
- Extraemos la implementación de las funciones en otros ficheros `.cpp`
- Normalmente, pondremos las funciones relacionadas en el mismo fichero `.cpp`
- Si queremos hacer una biblioteca (como veremos más adelante), se suele poner cada función en un fichero `.cpp`

iiiiiiii**IMPORTANTÍSIMO!!!!!!!**

Estos ficheros `.cpp` se van a compilar, por eso hay que incluir las bibliotecas que utilicen las funciones del fichero, por ejemplo `iostream`, o el espacio de nombre, etc.

División del código en ficheros

La distribución del programa tendría este aspecto



División del código en ficheros

Pregunta

¿Y por qué no extraer también los prototipos? ¿y los tipos de datos?

Ficheros de cabecera

Los prototipos de funciones y la definición de los tipos de datos se agrupan los **ficheros de cabecera**, que tienen extensión `.h`

Como el fichero principal, `main.cpp`, necesita los prototipos y los tipos de datos, se utiliza la directiva `include` para conectarlos.

Ejemplo

```
#include "funciones.h", que significa inserta aquí el contenido de funciones.h
```


División del código en ficheros

Extracción de los prototipos

- Se pueden crear tantos archivos de cabecera como se quiera para incluir los prototipos
- Normalmente, para las funciones de una biblioteca, se crea un fichero de cabecera que las recoja todas

Extracción de los tipos de datos

Para cada tipo de dato creado se crea un fichero de cabecera

Documentación

Los archivos del proyecto que se documentan son los `.h`

División del código en ficheros

La distribución del programa tendría este aspecto



División del código en ficheros

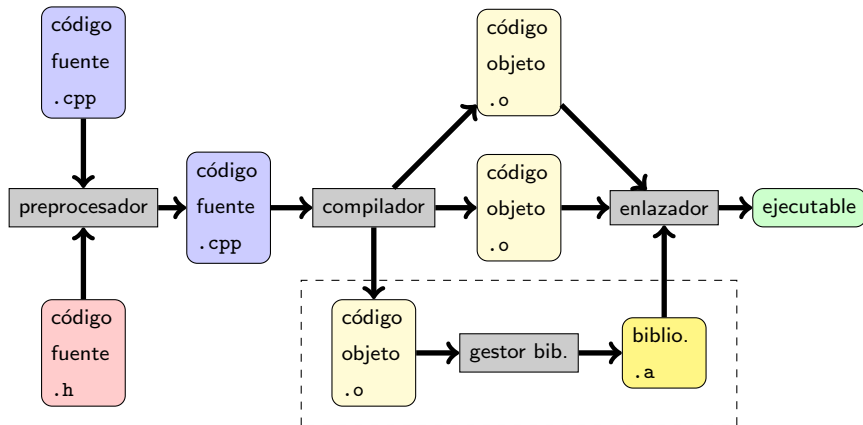
CUIDADO

En C++, todos los objetos deben ser declarados y definidos antes de ser usados. Por eso hay que tener cuidado donde se utiliza la directiva `include` en el fichero principal

CUIDADO

No hay una única manera eficiente de distribuir el código, depende del uso que queramos darle

Proceso de compilación



Construcción de bibliotecas

Proceso de compilación

Importante

- Los únicos ficheros que se compilan son `.cpp`
- La documentación del programa se realiza en los ficheros `.h`

Ocultamiento del software

Al dividir el código de esta forma podemos ocultar la implementación de un algoritmo al usuario:

- El usuario recibe el fichero compilado `.o` (ilegible para una persona) para que pueda utilizar el algoritmo
- El usuario recibe la documentación en el fichero de cabecera `.h` para que sepa cómo utilizarlo
- El fichero `.cpp` no es necesario para el usuario

Proceso de compilación

Normalmente, los archivos se distribuyen en subdirectorios para ordenarlos

Por ejemplo

- **src**, que contenga los ficheros .cpp
- **include**, que contenga los ficheros .h
- **obj**, que contenga los ficheros .o
- **lib**, que contenga los ficheros .a
- **bin**, que contenga los ficheros ejecutables
- **doc**, que contenga la documentación

El preprocesador

Preprocesador

El preprocesador es una herramienta que filtra el código fuente y lo prepara para ser compilado

- **Elimina los comentarios**
- **Procesa las directivas de preprocesamiento**, que son las que empiezan por #
Por ejemplo, `#include <iostream>`

- Recibe ficheros de texto (.cpp y .h)
- Devuelve fichero de texto (.cpp)
- En este proceso es cuando se utilizan los **ficheros de cabecera**

El preprocesador

Directiva include

La directiva `#include` hace que se incluya el contenido del fichero especificado en la posición en la que se encuentra la directiva

Ejemplo

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

Dos formas de especificar el fichero

- Entre `<` y `>`. Busca el fichero en el directorio de ficheros de cabecera del sistema o en el que se indique en la compilación
- Entre comillas. Busca el fichero en el directorio donde se realiza la compilación

El preprocesador

Fichero funciones.h

```
char bienvenida (int n);  
int contar (char p);
```

Fichero main.cpp

```
#include <iostream>  
#include "funciones.h"  
using namespace std;  
int main(){  
    int n=3;  
    cout << contar (bienvenida(n));  
}
```

include es lo mismo que *copiar y pegar*

El preprocesador

Directiva define

La directiva `#define` se emplea para definir constantes simbólicas
`#define identificador texto`*sustitución*

Ejemplo

```
#define MAXLONG 256
```

la directiva sustituye las apariciones de `MAXLONG` por `256`. Funciona como *buscar y reemplazar*

Ejemplo

```
#define CONST
```

define una constante `CONST` pero no sustituye nada, se utiliza para marcar si hemos realizado un proceso

El preprocesador

Directiva undef

La directiva `#undef` anula una definición previa

```
#undef identificador
```

Ejemplo

```
#undef MAXLONG
```

elimina la definición previa de `MAXLONG`, una vez hecho esto se puede asociar a la variable una nueva definición

Inclusión condicional de código

Existe la posibilidad de incluir código de forma condicional mediante las directivas `#if`, `#else`, `#elif`, `#endif`, `#ifdef`, `#ifndef`

Vamos a utilizar estas directivas para **evitar la doble inclusión de ficheros de cabecera**

El preprocesador

Fichero DNI.h

```
struct DNI{  
    char numero[8];  
    char letra;  
};
```

Fichero PERSONA.h

```
#include "DNI.h"  
  
struct PERSONA{  
    char nombre[45];  
    DNI documento;  
};
```

El preprocesador

Fichero main.cpp

```
#include "DNI.h"
#include "PERSONA.h"

int buscar (DNI m, PERSONA * m);

int main(){
    .....
    .....
};
```

El preprocesador

Fichero main.cpp (después del preprocesador)

```
struct DNI{  
    char numero[8];  
    char letra;  
};  
struct DNI{  
    char numero[8];  
    char letra;  
};  
struct PERSONA{  
    char nombre[45];  
    DNI documento;  
}; int buscar (DNI m, PERSONA * m);  
  
int main(){.....};
```

Doble definición del tipo de dato **DNI**

El preprocesador

Para evitar ésta transitividad, los ficheros de cabecera tendrán el siguiente formato

Fichero cabecera.h

```
#ifndef CABECERA_H // si no se ha incluido, es decir,  
                  // si no se ha definido la constante  
#define CABECERA_H // inclúyelo, es decir, define la constante  
.  
.  
. // contenido del fichero de cabecera  
.  
.  
#endif // final del condicional
```

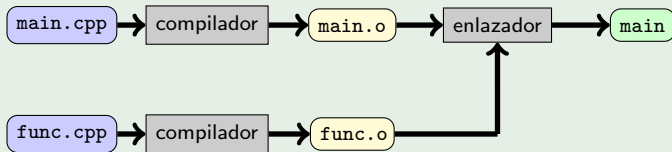

El enlazador

Enlazador

el enlazador tiene como objetivo resolver las referencias a objetos externos, es decir, realizar las llamadas a objetos contenidos en otros ficheros previamente compilados

Diagrama

Supongamos que `main.cpp` hace una llamada a una función en `func.cpp`



Comentarios

- En este proceso es cuando se utilizan las bibliotecas
- El enlazador enlaza ficheros .o, en particular, los contenidos en una biblioteca
- No haremos una llamada al enlazador, se llamara al compilador para que también enlace
- Para que el enlazador construya el fichero ejecutable es necesario que, entre los ficheros .o, uno contenga la función `main`

Sintaxis del coompilador g++

g++

g++ es el compilador de C++ de GNU

Es también el compilador por defecto que usa Dev-Cpp de Windows o Xcode de macOS

Sintaxis general desde la linea de comandos

g++ -[opción] [argumentos] nombre_fichero

- Todas las opciones vienen precedidas de -
- Algunas opciones no tienen argumentos
- nombre_fichero es el fichero a procesar

Sintaxis del compilador g++

Opción -o

`-o fichero_salida`

- sirve para especificar el nombre del fichero de salida del proceso de compilación
- si no se especifica el fichero de salida, entonces toma el nombre:
 - `a.exe` (DOS) `a.out` (LINUX) para ficheros ejecutables
 - `nombre_fichero.o` si sólo se construye el objeto

Ejemplo

```
g++ -o juego main.cpp
```

```
g++ -o juego.exe main.cpp
```

Sintaxis del compilador g++

Opción -c

-c

- no tiene argumentos
- sirve para realizar la tarea de preprocesamiento y la compilación, pero no el proceso de enlazado
- el fichero de salida es un fichero .o
- es necesario utilizar esta opción al compilar ficheros .cpp que no contienen a la función main

Ejemplo

```
g++ -c subir.cpp
```

```
g++ -c -o funcion.o funcion.cpp
```

Sintaxis del compilador g++

Opción -I

-I camino

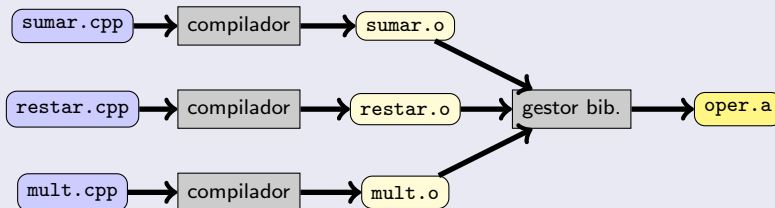
- añade camino a la lista de directorios donde se buscan ficheros de cabecera
- si no se utiliza, sólo busca en los directorios del sistema y en el directorio donde se realiza la compilación
- es posible utilizarlo varias veces para añadir varios directorios

Ejemplo

```
g++ -c subir.cpp -I /include  
g++ -c -o func.o func.cpp -I /include -I /otroinc
```


Construcción de bibliotecas

Proceso de creación de una biblioteca



Construcción de bibliotecas

Nota

Cada módulo de una biblioteca es un fichero `.o` que puede contener más de una función. Pero es más eficiente, en espacio, que cada función se implemente en un fichero `.cpp` distinto

Ejemplo

Biblioteca `libmov.a` con dos funciones: `subir` y `bajar`

- Modo 1: `libmov.a` con un módulo: `subirbajar.o`
- Modo 2: `libmov.a` con dos módulos: `subir.o` y `bajar.o`

Supongamos que `main` sólo utiliza la función `subir`

- Tamaño ejecutable modo 1: 5234 bytes
- Tamaño ejecutable modo 2: 3136 bytes

!!!La biblioteca sólo utiliza los módulos que necesita!!!

Resumen: modo de construcción de una biblioteca

- 1 Implementar las funciones de la biblioteca
- 2 Cada función en un fichero `.cpp` independiente (más o menos)
- 3 Compilar los ficheros `.cpp` para obtener los módulos `.o`
- 4 Construir el fichero de la biblioteca `.a`
- 5 Implementar un fichero de cabecera `.h` con la declaración de las funciones y su documentación
- 6 Dar al usuario el fichero de cabecera `.h` y el de biblioteca `.a`

Construcción de bibliotecas

Gestor de bibliotecas ar

ar es el gestor de bibliotecas de GNU

Sintaxis de ar

```
ar -[opción] [modificadores] biblioteca [ficheros_objeto]
```

- Opción r, **añade**, o reemplaza si ya existía alguno, los ficheros objeto en la biblioteca. Si no existe la biblioteca, la crea
- Opción d, **elimina** los ficheros objeto de la biblioteca
- Opción x, **extrae** el fichero objeto de la biblioteca
- Opción t, **lista** los ficheros objeto en la biblioteca
- Modificador s, **crea** un índice de los ficheros objeto, necesario para el enlazado
- Modificador v, **muestra** las operaciones realizadas

Sintaxis del compilador g++

Opción -L

-L camino

- añade camino a la lista de directorios donde se buscan ficheros de biblioteca
- si no se utiliza, sólo busca en los directorios del sistema y en el directorio donde se realiza la compilación
- es posible utilizarlo varias veces para añadir varios directorios

Ejemplo

```
g++ -o juego juego.cpp -L /lib
```

```
g++ -o juego juego.cpp -L /lib -I /graficos/lib
```

Opción -l

`-lnombre_biblioteca`

- señala al compilador que utilice los módulos de la biblioteca `libnombre_biblioteca.a`
- es posible utilizarlo varias veces para añadir varias bibliotecas

Ejemplo

```
g++ -o juego juego.cpp -L /lib -loper
```

```
g++ -o juego juego.cpp -L /lib -loper -lmenu
```

Ficheros makefile

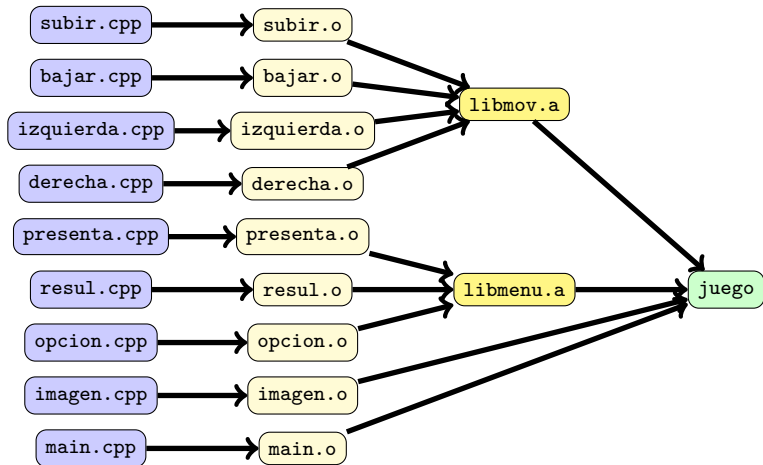
Supongamos que queremos realizar un proyecto que cuenta con los siguientes ficheros de código fuente `.cpp`:

- `subir.cpp`, `bajar.cpp`, `izquierda.cpp`, `derecha.cpp` conteniendo funciones para crear una biblioteca `libmov.a`
- `presenta.cpp`, `resul.cpp`, `opcion.cpp` conteniendo funciones para crear una biblioteca `libmenu.a`
- `imagen.cpp` conteniendo la implementación de la clase `imagen`
- `main.cpp` con la función `main`

Ejercicio

¿Qué ordenes hacen falta para compilar todo el proyecto?

Motivación



Solución

```
g++ -c subir.cpp
g++ -c bajar.cpp
g++ -c izquierda.cpp
g++ -c derecha.cpp
ar -rsv libmov.a subir.o bajar.o izquierda.o derecha.o
g++ -c presenta.cpp
g++ -c resul.cpp
g++ -c opcion.cpp
ar -rsv libmenu.a presenta.o resul.o opcion.o
g++ -c imagen.cpp
g++ -c main.cpp
g++ -o juego main.o imagen.o -lmov -lmenu
```

Problema

Al realizar algún cambio, necesitaremos compilar parte del proyecto

Ejemplo

Si se realizan cambios en `imagen.cpp`...

```
g++ -c imagen.cpp
```

```
g++ -o juego main.o imagen.o -lmov -lmenu
```

...y en todas las funciones que utilicen la clase...

...y las bibliotecas de las que forman parte...

Si el proyecto es grande, **compilar así es poco eficiente**

Motivación

Una solución

Guardar las ordenes en un fichero

Ficheros makefile

Un fichero `makefile` es un fichero de texto donde se guardan las ordenes de compilación de un proyecto

Programa make

El programa `make` es una aplicación que lee ficheros `makefile` y ejecuta las ordenes escritas en él

Aquí trataremos el programa `make` de GNU y la construcción de ficheros `makefile` para ser leídos por él

Sintaxis básica

- Los comentarios van precedidos de #
- Cada acción que queramos realizar tiene el formato

```
destino : [dependencias]
```

```
[tabulador]orden1
```

```
[tabulador]orden2
```

```
[tabulador]orden3
```

```
.....
```

```
[tabulador]ordenM
```

- destino es un identificador de la lista de ordenes a realizar
- las dependencias son otros destinos que hacen falta para realizar destino

Ficheros makefile

```
*makefile
1 # makefile algo rústico para compilar el proyecto anterior
2
3 compilar:
4     g++ -c subir.cpp
5     g++ -c bajar.cpp
6     g++ -c izquierda.cpp
7     g++ -c derecha.cpp
8     ar -rsv libmov.a subir.o bajar.o izquierda.o derecha.o
9     g++ -c presenta.cpp
10    g++ -c resul.cpp
11    g++ -c opcion.cpp
12    ar -rsv libmenu.a presenta.o resul.o opcion.o
13    g++ -c imagen.cpp
14    g++ -c main.cpp
15    g++ -o juego main.o imagen.o -lmov -lmenu
16
```

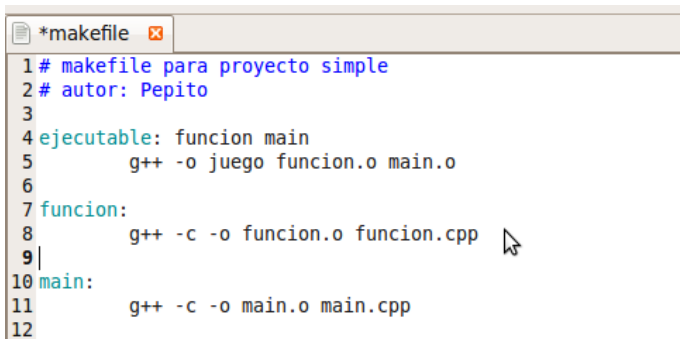
Ficheros makefile

Normalmente se divide en procesos más pequeños,

```
*makefile
1 # makefile algo menos r ístico
2         u
3 ejecutable: imagen mov menu
4     g++ -c main.cpp
5     g++ -o juego main.o imagen.o -lmov -lmenu
6 mov:
7     g++ -c subir.cpp
8     g++ -c bajar.cpp
9     g++ -c izquierda.cpp
10    g++ -c derecha.cpp
11    ar -rsv libmov.a subir.o bajar.o izquierda.o derecha.o
12 menu:
13    g++ -c presenta.cpp
14    g++ -c resul.cpp
15    g++ -c opcion.cpp
16    ar -rsv libmenu.a presenta.o resul.o opcion.o
17 imagen:
18    g++ -c imagen.cpp
19 |
```

Ficheros makefile

Otro ejemplo,



```
1 # makefile para proyecto simple
2 # autor: Pepito
3
4 ejecutable: funcion main
5     g++ -o juego funcion.o main.o
6
7 funcion:
8     g++ -c -o funcion.o funcion.cpp
9
10 main:
11     g++ -c -o main.o main.cpp
12
```


Sintaxis del programa make

`make -[opciones] [destinos]`

- `destinos` son los destinos que queremos que se realicen del fichero `makefile`
- Si no añadimos ningún destino, intentará realizar sólo el primero y los destinos de los que dependa
- Si falla al construir un destino, se detiene el proceso y borra el destino
- Opciones:
 - `-h`, muestra la ayuda del programa
 - `-f nom_fich`, indica que el fichero `makefile` tiene por nombre `nom_fich`. Si no se indica, buscará un fichero con nombre `makefile` o `Makefile`
 - `-n` muestra las instrucciones que se ejecutarían, sin ejecutarlas

Ficheros makefile

Ejemplos de lo mismo

```
make  
make -f makefile  
make -f Makefile
```

Ejemplos

```
make -f makefil.mak imagen.o  
make -f makefil.txt imagen.o mov  
make mov menu  
make -n menu  
make -h
```

Propiedad

SÓLO si el fichero destino es más reciente que el fichero fuente, el programa `make` reconstruirá el fichero destino

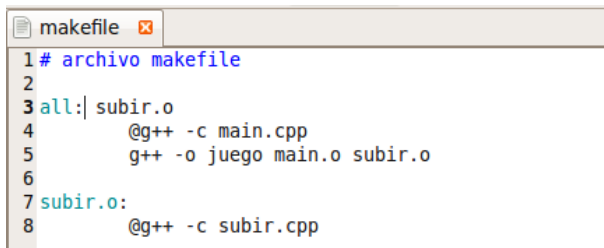
Ficheros makefile

Prefijos en las ordenes de un fichero makefile

Prefijo @

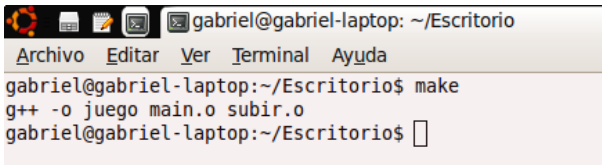
@g++ -c subir.cpp

No muestra la orden por pantalla, pero sí la ejecuta



```
makefile x
1 # archivo makefile
2
3 all:| subir.o
4     @g++ -c main.cpp
5     g++ -o juego main.o subir.o
6
7 subir.o:
8     @g++ -c subir.cpp
```

Ficheros makefile



A terminal window titled 'gabriel@gabriel-laptop: ~/Escritorio'. The window has a menu bar with 'Archivo', 'Editar', 'Ver', 'Terminal', and 'Ayuda'. The terminal shows the following commands and output:

```
gabriel@gabriel-laptop:~/Escritorio$ make
g++ -o juego main.o subir.o
gabriel@gabriel-laptop:~/Escritorio$
```

Prefijos en las ordenes de un fichero makefile

Prefijo -


`-g++ -c subir.cpp`

Ignora los errores producidos por la orden

Destinos simbólicos

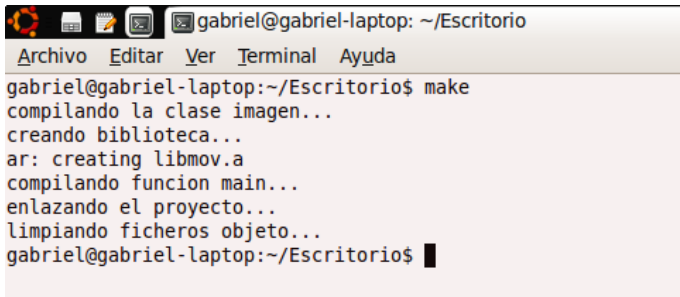
- Un destino simbólico es un destino que agrupa varios destinos diferentes
- No tiene ordenes y la lista de dependencias son los destinos a agrupar
- Se realizan para simplificar la llamada al programa `make`

Ficheros makefile



```
1 # archivo makefile
2
3 all: proyecto limpiar|
4
5 proyecto: imagen.o libmov.a main.o
6         @echo enlazando el proyecto...
7         @g++ -o juego main.o subir.o
8
9 imagen.o:
10        @echo compilando la clase imagen...
11        @g++ -c imagen.cpp
12
13 libmov.a:
14        @g++ -c subir.cpp
15        @g++ -c bajar.cpp
16        @echo creando biblioteca...
17        @ar -rs libmov.a subir.o bajar.o
18
19 main.o:
20        @echo compilando funcion main...
21        @g++ -c main.cpp
22
23 limpiar:
24        @echo limpiando ficheros objeto...
25        @rm *.o
```

Ficheros makefile



A terminal window titled 'gabriel@gabriel-laptop: ~/Escritorio' with a menu bar containing 'Archivo', 'Editar', 'Ver', 'Terminal', and 'Ayuda'. The terminal shows the following output after running 'make':

```
gabriel@gabriel-laptop:~/Escritorio$ make
compilando la clase imagen...
creando biblioteca...
ar: creating libmov.a
compilando funcion main...
enlazando el proyecto...
limpiando ficheros objeto...
gabriel@gabriel-laptop:~/Escritorio$
```


Macros

Una macro es una cadena que se expande cuando se llama desde un fichero makefile

Sintaxis

`NOMBRE = cadena`

- `NOMBRE` es el identificador de la macro. Sin espacios en blanco. Suele escribirse en mayúsculas
- `cadena` es la cadena a expandir

La llamada a la macro se hace de la forma `$(NOMBRE)`

Ficheros makefile

```
*makefile
1 # archivo makefile
2
3 objetos = subir.o bajar.o
4 obj = obj/
5
6 all: proyecto limpiar
7 proyecto: imagen.o libmov.a main.o
8     @echo enlazando el proyecto...
9     @g++ -o juego $(obj)main.o -lmov
10 imagen.o:
11     @echo compilando la clase imagen...
12     @g++ -c $(obj)imagen.o imagen.cpp
13 libmov.a:
14     @g++ -c $(obj)subir.o subir.cpp
15     @g++ -c $(obj)bajar.o bajar.cpp
16     @echo creando biblioteca...
17     @ar -rs libmov.a $(obj)$(objetos)
18 main.o:
19     @echo compilando funcion main...
20     @g++ -c $(obj)main.o main.cpp
21 limpiar:
22     @echo limpiando ficheros objeto...
23     @rm $(obj)*.o
```