Práctica 2: Abstracción

Makefiles, Templates & Otros

Gustavo Rivas Gervilla

UNIVERSIDAD DE GRANADA

Contenido

Makefiles

- En la práctica vamos a trabajar con proyectos grandes, con diversos ficheros y con unos procesos de compilación más complejos.
- Automatizar esta tarea nos permite ahorrar tiempo y errores.
- Un makefile nos puede ayudar a conseguirlo.[1]

[1] Alex Allain. *Makefiles*. https://www.cprogramming.com/tutorial/makefiles.html. [Online; accedido 14/10/2019]; Tutorialspoint. *Makefile Tutorial*. https://www.tutorialspoint.com/makefile/index.htm. [Online; accedido 14/10/19].

Makefiles: targets

Un archivo makefile está compuesto por unos targets, que pueden tener dependencias de otros targets o archivos, y desencadenan la ejecución de un conjunto de instrucciones:

```
1  $(LIB)/libracional.a : $(OBJ)/Racional.o
2          ar rvs $(LIB)/libracional.a $(OBJ)/Racional.o
3
4  $(OBJ)/Racional.o : $(SRC)/Racional.cpp
5          $(CXX) $(CPPFLAGS)  -o $(OBJ)/Racional.o $(SRC
               )/Racional.cpp -I$(INCLUDE)
```

Por lo tanto podemos usar makefiles para todo tipo de tareas que puedan expresarse de este modo (objetivos, dependencias e instrucciones), y no sólo para compilar proyectos escritos en C++.

Makefiles: macros

- Podemos definir macros (que las podemos ver como variables) y luego acceder a su valor con `$(nombreMacro)$`.

```
1  INCLUDE = include
2  LIB = lib
3  OBJ = obj
4  SRC = src
```

- Los comentarios comienzan con `#` y podemos incluir comentarios de varias líneas usando `\`.

Makefiles: macros

Algunas de estas macros ya están predefinidas:

`$@`	El nombre del target (del archivo a generar).
`$?`	Los nombres de las dependencias que han cambiado.
`$<`	La primera dependencia.
`$*`	El prefijo compartido por un target y los archivos de los que depende.

```
1  hello: main.cpp hello.cpp factorial.cpp
2          $(CC) $(CFLAGS) $? $(LDFLAGS) -o $@
3
4  .cpp.o:
5          $(CC) $(CFLAGS) -c $<
6
7  #O alternativamente:
8
9  .cpp.o:
10         $(CC) $(CFLAGS) -c $*.c
```

Hay otras muchas, y las podemos ver con `make -p`.

Makefiles: directivas

Podemos usar sentencias condicionales en nuestros makefiles:

```
1  libs_for_gcc = -lgnu
2  normal_libs =
3
4  foo: $(objects)
5  ifeq ($(CC),gcc)
6          $(CC) -o foo $(objects) $(libs_for_gcc)
7  else
8          $(CC) -o foo $(objects) $(normal_libs)
9  endif
```

También tenemos la sentencia `include`.

Makefiles

- Los objetivos pueden estar en cualquier orden.
- Por defecto se ejecuta el primer objetivo que aparece en el archivo.
- Cada instrucción ha de estar precedida por un tabulador.
- Podemos ignorar el estado devuelto por un comando (si hubo error o no), anteponiendo - al comando.

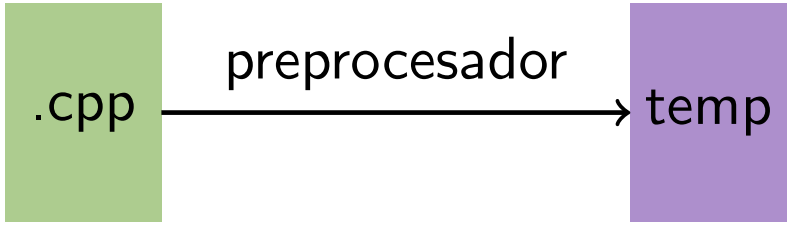
```
1  clean:
2          -rm -f *.o
3
4  rebuild: clean build
```

Makefiles: El Proceso de Compilación

Una vez hemos entendido cómo funciona un makefile vamos a usarlo para compilar un proyecto escrito en C++. Y para saber cómo escribir un makefile para compilar este tipo de proyectos lo mejor es conocer el proceso de compilación de C++[2].

[2] Kurt McMahon. *The C++ compilation process*. http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html. [Online; accedido 14/10/2019]; Daniel Ángel Muñoz Trejo. *How C++ Works: Understanding Compilation*. https://www.toptal.com/c-plus-plus/c-plus-plus-understanding-compilation. [Online; accedido 14/10/2019].

Makefiles: El Proceso de Compilación

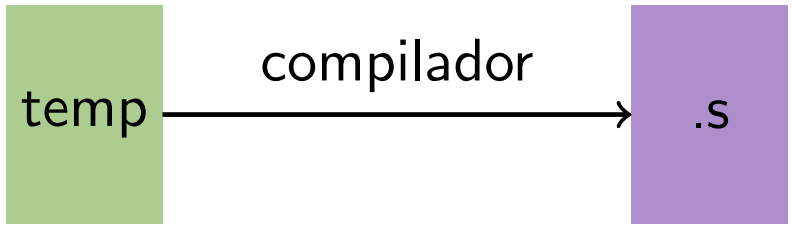


- Se traducen los `#include` por el código del fichero.
- Elimina algunos trozos de código si no se han de compilar por alguna condición de compilación evaluada a `false`.
- Reemplaza macros como cuando hacemos `#define TABLE_SIZE 200` y luego usamos `TABLE_SIZE` como el tamaño con el que declarar un array.

```
g++ -E foo.cpp -o foo.ii
```

Con la opción `-E` paramos el proceso después de la acción del preprocesador, obteniendo así el archivo generado por él.

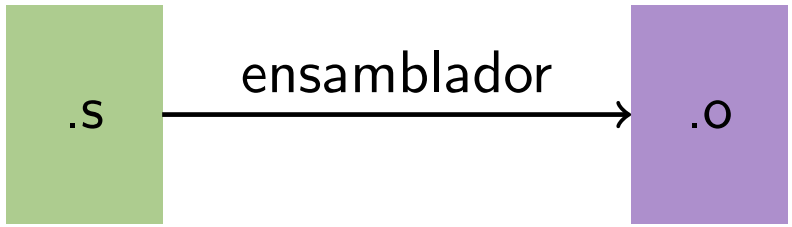
Makefiles: El Proceso de Compilación



- Traducimos el código expandido a código ensamblador.
- Podemos parar el proceso de compilación tras la acción del compilador con la opción `-S`:

```
g++ -S foo.cpp -o foo.ss
```

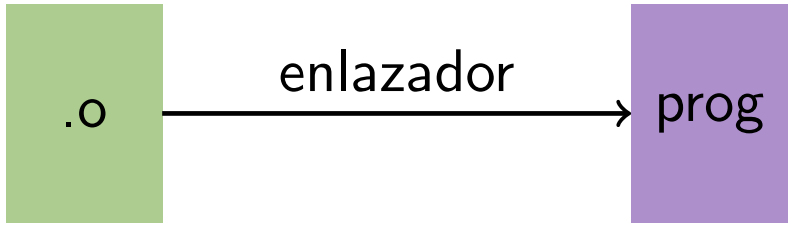
Makefiles: El Proceso de Compilación



- Se pasa el código ensamblador a código objeto.
- Podemos parar el proceso de compilación tras la acción del ensamblador con la opción `-c`:

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

Makefiles: El Proceso de Compilación



- Ahora hemos generado código objeto que puede contener símbolos no definidos.
- Por lo tanto tenemos que enlazar esos códigos objeto entre sí para que no haya referencias perdidas y que por tanto el enlazador nos devuelva el error `missing symbol`.
- Podemos unir varios `.o` en una sola librería, podemos ver las liberías como un `.zip`[3] de códigos objeto, así sólo tenemos que enlazar las librería con todos los `.o` que necesita otro `.o`.

```
g++ -o foo foo.o -lbar
```

[3] https://stackoverflow.com/a/1907856/5636497

Contenido

Templates

Los templates son una herramienta de generalización fundamental: facilita el mantenimiento del código y disminuye el trabajo requerido para desarrollarlo.

```
1  class Test<T, U>
2  {
3      T obj1;  // An object of type T
4      U obj2;  // An object of type U
5
6      Test(T obj1, U obj2) // constructor
7      {
8          this.obj1 = obj1;
9          this.obj2 = obj2;
10     }
11
12     public void print() // To print objects of T and U
13     {
14         System.out.println(obj1);
15         System.out.println(obj2);
16     }
17 }
```

Código Java para implementar una clase genérica.

Templates: Opciones de Implementación

Podemos distinguir tres mecanismos con los que trabajar con `templates` en C++[4]:

1. Inclusión de las definiciones.
   - Hacer `#include "vector_dinamico.cpp"` en el `vector_dinamico.h`.
   - Inconvenientes: dejamos de ocultar información y además generamos un código ejecutable de mayor tamaño.
2. Instanciación implícita.
   - Modularizamos de la forma usual e incluimos en el `.cpp` las instanciaciones que necesitemos, por ejemplo `template vector_dinamico<float>;`.
   - Inconveniente: tenemos que introducir a mano las instanciaciones que necesitemos, aumentando el trabajo en el mantenimiento del código.

[4] Carrillo y Fernández-Valdivia 2006.

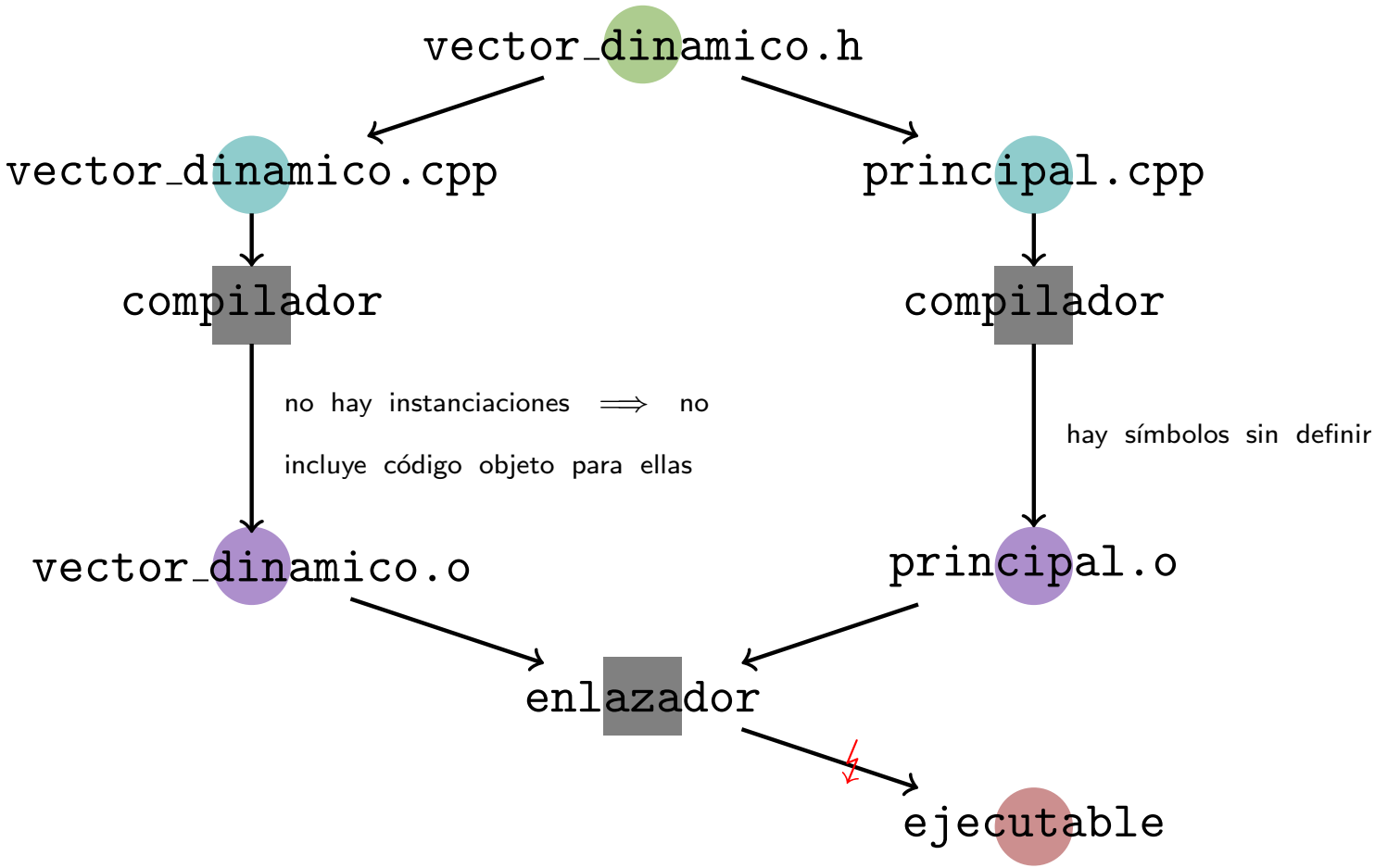
Template: Opciones de Implementación

3. Uso de la palabra reservada `export`.
   - Inconveniente: esta palabra reservada ya no se contempla en el estándar[5].

[5] isocpp. *Templates*. https://isocpp.org/wiki/faq/templates#templates-defn-vs-decl. [Online; accedido 14/10/2019]; cppreference. *C++ keywords: export*. https://en.cppreference.com/w/cpp/keyword/export. [Online; accedido 14/10/2019].

Templates: Compilación

Trabajar con `templates` en C++ cambia el modo en el que tenemos que organizar nuestro código para compilarlo correctamente[6]:



[6] Antonio Garrido Carrillo y Joaquín Fernández-Valdivia (2006). *Abstracción y estructuras de datos en C++*. Delta Publicaciones.

Contenido

Reserva de Memoria

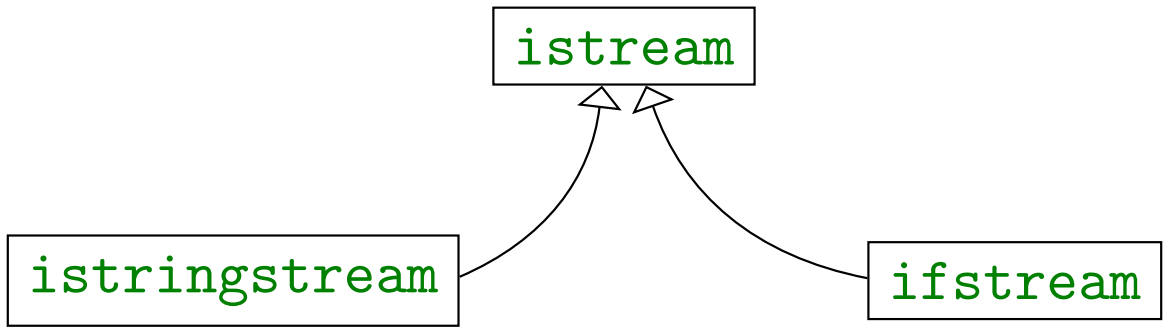
- La reserva de memoria (`new[]` y `delete[]`) $\Longrightarrow$ `memory leaks`.

```
valgrind --leak-check=full ./bin/prueba datos/
                                    prueba.txt
```

- Para analizar el uso de memoria que hace nuestro programa podemos usar `valgrind`.
- Por otro lado en Java y otros lenguajes tenemos lo que se conoce como `garbage collector` .

E/S con Ficheros en C++

En la práctica necesitaremos cargar los ingredientes y otros datos desde archivos `.txt`. Para trabajar con fichero C++ proporciona las siguientes clases:



- Tenemos métodos como (entre otros de igual utilidad):
  - `get`
  - `getline`
  - `peek`
  - `ignore`
  - `good`

Contenido

Información

- ☐ El nombre del autor o autores está en el/los archivo/s entregados: incluyendo cada `.h` y `.cpp`.
- ☐ Cada uno de los autores de la práctica ha subido la entrega a PRADO en la fecha acordada.
- ☐ Todos los autores de la práctica han entregado los mismos archivos.
- ☐ Se sigue la estructura de ficheros propuesta.
- ☐ Se implementan todos los métodos/códigos requeridos en el guión.
- ☐ El código y la documentación compilan correctamente ∴ PDF explicativo.

¡Buena semana!