Metodología de la Programación

MODULARIZACIÓN

2. GESTIÓN DE PROYECTOS CON `make` Y FICHEROS `makefile`

Francisco J. Cortijo Bon
cb@decsai.ugr.es

Curso 2021-2022

Resumen

La gestión y mantenimiento del software durante el proceso de desarrollo puede ser una tarea ardua si trabajamos con diferentes ficheros fuente y se utilizan, además, funciones ya incorporadas en ficheros de biblioteca.

Durante el proceso de desarrollo se modifica frecuentemente el software y las modificaciones incorporadas pueden afectar a otros módulos. Estas modificaciones deben propagarse a los módulos que dependen de aquellos que han sido modificados generando una cascada de modificaciones. El objetivo es que el programa ejecutable final refleje las modificaciones introducidas

Saber qué módulos hay que actualizar, el orden en que se deben realizar las actualizaciones y proceder a realizarlas puede ser una tarea larga y compleja. En este documento veremos cómo el programa `make` (basándose en ficheros `makefile`) se puede encargar de la gestión automatizada de los proyectos.

Índice

1. Introducción

En la primera entrega de este seminario de modularización aprendimos a modularizar un programa en diferentes ficheros. Se trataba de casos sencillos en los que un ejecutable se construye a partir de un fichero fuente que contiene la función `main` y módulos. Cada uno tiene asociado dos ficheros de código fuente:

1. Un **fichero de cabecera** (extensión `.h`) con la **declaración** de las funciones.
2. Un **fichero de código fuente** (extensión `.cpp`) con la **definición** de las funciones.

Decíamos que en el contexto de esta asignatura, un **proyecto** software complejo se compone de diferentes módulos de código que se *combinan* para generar ejecutables.

Durante el proceso de desarrollo se modifica frecuentemente el software y las modificaciones incorporadas pueden afectar a otros módulos. Por ejemplo, cualquier cambio en una función de un módulo de funciones obligará a reconstruir el fichero objeto que se construyó a partir de la versión anterior, y en consecuencia también habría que actualizar el ejecutable que se construyó a partir de la versión anterior del módulo objeto.

La gestión y mantenimiento del software durante el proceso de desarrollo puede ser una tarea ardua si éste se estructura en diferentes ficheros fuente y se utilizan, además, funciones ya incorporadas en ficheros de biblioteca.

Estas modificaciones deben propagarse a los módulos que dependen de aquellos que han sido modificados generando una cascada ingente de modificaciones. El objetivo es que el programa ejecutable refleje las modificaciones introducidas.

En proyectos complejos la gestión “manual” es imposible.

1. Se requiere una herramienta para la gestión automática que *analice* las dependencias entre los módulos y *decida* qué ficheros debe reconstruir.

   Hablamos del programa `make`.

2. Se necesita especificar las *dependencias* entre módulos y las *tareas* que deben hacerse cuando se detecta que hay que reconstruir algún módulo

   Hablamos de ficheros *makefile*.

Suponemos que conoce `make` y lo ha practicado así como con ficheros *makefile*.

2. Proyecto Kaprekar

El proyecto que vamos a gestionar consiste en la creación de un ejecutable que soluciona el problema del **Proceso de Kaprekar**. Realmente el proyecto será capaz de generar dos ejecutables: uno basado en la solución no modularizada y otro en la modularizada.

Solución no modularizada.

La descripción de este problema y la solución no modularizada está disponible en PRADO en el fichero `Kaprekar.cpp`. El proyecto que vamos a desarrollar incorporará la tarea de generar el ejecutable a partir de la solución no modualrizada. El diagrama de dependencias para la generación de este ejecutable (con indicación del orden) será el que se indica en la figura 1.

Figura 1: Diagrama de dependencias. Solución no modular

Recuerden cómo se creaba el ejecutable en dos pasos:

1. `g++ -c -o obj/Kaprekar.o src/Kaprekar.cpp`
2. `g++ -o bin/Kaprekar obj/Kaprekar.o`

Solución modularizada.

La solución modularizada se explicó en la primera entrega de este seminario. Vamos a recordarla rápidamente. La estructura del proyecto (diagrama de dependencias) que nos ocupa ahora, resultado de la modularización será el que se indica en la figura 2. Recuerde:

- `main_Kaprekar.cpp` contendrá la función `main` únicamente.
- `funciones_Kaprekar.h` contendrá las **declaraciones** de las funciones.
- `funciones_Kaprekar.cpp` contendrá las **definiciones** de las funciones.

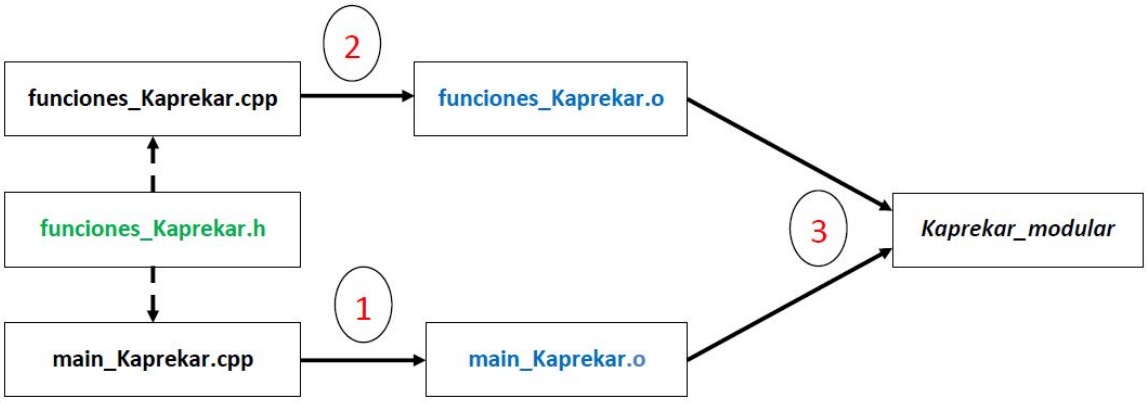

Figura 2: Diagrama de dependencias del proyecto completo

Para la generación del ejecutable nos basamos en el esquema de dependencias del proyecto (figura 2). Observe que los pasos 1 y 2 son intercambiables.

1. `g++ -c -o obj/main_Kaprekar.o src/main_Kaprekar.cpp -I./include`
2. `g++ -c -o obj/funciones_Kaprekar.o src/funciones_Kaprekar.cpp -I./include`
3. `g++ -o bin/Kaprekar_modular.o obj/main_Kaprekar.o obj/funciones_Kaprekar.o`

3. Fichero `makefile`

El fichero `makefile` asociado al proyecto será, en este momento, bastante simple. Distinguiremos las secciones *lógicas*:

1. **Cabecera**. Son comentarios informativos sobre la autoría y propósito del proyecto.
2. **Macros**. Sirven para particularizar la localización de las carpetas con los ficheros del proyecto.
3. **Destino simbólico `all`**. Se especifican los destinos o ficheros a generar.
4. **Reglas e instrucciones para generar ejecutables**.
5. **Reglas e instrucciones para generar objetos**.
6. **Limpieza**.

1. ***Cabecera***

```
#################################################################
#
# METODOLOGIA DE LA PROGRAMACION
#
# (C) FRANCISCO JOSE CORTIJO BON
# DEPTO. DE CIENCIAS DE LA COMPUTACION E INTELIGENCIA ARTIFICIAL
#
# fichero: makefile
#
# makefile para el problema de Kaprekar.
#
#################################################################
```

2. ***Macros***

Supondremos que existe la estructura de carpetas habitual `src`, `include`, etc. Todas ellas son descendientes directas de la carpeta actual, donde reside el fichero `makefile`.

```
HOME = .
BIN = $(HOME)/bin
SRC = $(HOME)/src
OBJ = $(HOME)/obj
LIB = $(HOME)/lib
INCLUDE = $(HOME)/include
```

3. *Destino simbólico* `all`

En este proyecto vamos a realizar lo básico, así que indicaremos que queremos crear los dos ejecutables. Cuidado con indicar la carpeta donde van a guardarse.

Observe cómo añadimos destinos simbólicos (`preambulo`, `ejecutables` y `fin-ejecutables`) para que la ejecución de `make` muestre mensajes de guía en el terminal.

```
all: \
    preambulo \
    ejecutables \
    $(BIN)/Kaprekar \
    $(BIN)/Kaprekar_modular \
    fin-ejecutables
```

Los destinos simbólicos adicionales, y las instrucciones que se ejecutan son los siguientes. Observen que todas las instrucciones son `echo`.

```
preambulo:
    @echo
    @echo ------------------------------------------------------------
    @echo METODOLOGÍA DE LA PROGRAMACIÓN
    @echo
    @echo "("c")" Francisco José CORTIJO BON
    @echo Depto. Ciencias de la Computación e Inteligencia Artificial
    @echo Universidad de Granada
    @echo ------------------------------------------------------------
    @echo

ejecutables:
    @echo
    @echo Creando ejecutables...

fin-ejecutables:
    @echo
    @echo ...Ejecutables creados
    @echo
```

4. *Reglas e instrucciones para generar ejecutables*

```
#Figura 1. Regla 2
$(BIN)/Kaprekar : $(OBJ)/Kaprekar.o
    @echo
    @echo Creando ejecutable: Kaprekar
    @echo
    g++ -o $(BIN)/Kaprekar $(OBJ)/Kaprekar.o

#Figura 2. Regla 3
$(BIN)/Kaprekar_modular : $(OBJ)/main_Kaprekar.o \
                          $(OBJ)/funciones_Kaprekar.o
    @echo
    @echo Creando ejecutable: Kaprekar_modular
    @echo
    g++ -o $(BIN)/Kaprekar_modular \
           $(OBJ)/main_Kaprekar.o $(OBJ)/funciones_Kaprekar.o
```

5. *Reglas e instrucciones para generar objetos*

```
#Figura 1. Regla 1
$(OBJ)/Kaprekar.o : $(SRC)/Kaprekar.cpp
    @echo
    @echo Creando objeto: Kaprekar.o
    @echo
    g++ -c -o $(OBJ)/Kaprekar.o $(SRC)/Kaprekar.cpp

#Figura 2. Regla 1
$(OBJ)/main_Kaprekar.o : $(SRC)/main_Kaprekar.cpp \
                         $(INCLUDE)/funciones_Kaprekar.h
    @echo
    @echo Creando objeto: main_Kaprekar.o
    @echo
    g++ -c -o $(OBJ)/main_Kaprekar.o $(SRC)/main_Kaprekar.cpp \
            -I$(INCLUDE)

#Figura 2. Regla 2
$(OBJ)/funciones_Kaprekar.o : $(SRC)/funciones_Kaprekar.cpp \
                              $(INCLUDE)/funciones_Kaprekar.h
    @echo
    @echo Creando objeto: funciones_Kaprekar.o
    @echo
    g++ -c -o $(OBJ)/funciones_Kaprekar.o \
              $(SRC)/funciones_Kaprekar.cpp -I$(INCLUDE)
```

6. *Limpieza*

```
clean: clean-objs

clean-objs:
    @echo Borrando objetos...

    -rm $(OBJ)/Kaprekar.o
    -rm $(OBJ)/main_Kaprekar.o
    -rm $(OBJ)/funciones_Kaprekar.o

    @echo ...Borrados
    @echo

clean-bins :
    @echo Borrando ejecutables...

    -rm $(BIN)/Kaprekar
    -rm $(BIN)/Kaprekar_modular

    @echo ...Borrados
    @echo

mr.proper:  clean-objs clean-bins
```

4. Actividades

Deberá disponer de los ficheros:

- `Kaprekar.cpp` contendrá la solución no modularizada.
- `main_Kaprekar.cpp` contendrá la función `main` únicamente.
- `funciones_Kaprekar.h` contendrá las **declaraciones** de las funciones.
- `funciones_Kaprekar.cpp` contendrá las **definiciones** de las funciones.

`Kaprekar.cpp` está disponible en la primer entrega y los demás ficheros (`main_Kaprekar.cpp`, `funciones_Kaprekar.h` y `funciones_Kaprekar.cpp`) deben haber sido construidos siguiendo las indicaciones de la primera entrega.

Construya el fichero `makefile`. Hágalo siguiendo el orden indicado en la sección 3.

Durante las siguientes pruebas es posible que tenga que reconstruir el proyecto ejecutando `make`. Debería saber cuándo es necesario.

1. Haga que `make` realice la limpieza de los ficheros objeto (únicamente).
2. Inténtelo de nuevo. ¿Se produce algún error? ¿Se trata de un error no controlado que provoca la finalización de `make` o por el contrario `make` continua su ejecución procesando el resto del fichero `makefile`?
3. Reconstruya el proyecto. Después, sin modificar el fichero `makefile`, haga que `make` borre **todos** los ficheros generados
4. Modifique el fichero `makefile` para que, después de generar los ejecutables, borre los objetos.
5. Modifique el fichero `makefile` para que, antes de crear nada, borre los ejecutables y los objetos. ¿Qué sentido tiene esta modificación? ¿Cuáles son las ventajas e inconvenientes?
6. Cambie el orden de las reglas (el que deseeen) sin cambiar el destino simbólico `all`, que seguirá siendo el primero. Comprueben que todo se comporta igual.
7. Sin cambiarse de carpeta, **mueva** las carpetas `src`, `include`, `obj`, `lib` y `bin` a otra carpeta, pero deje el fichero `makefile` donde está. Modifique el fichero `makefile` (emplee las mínimas modificaciones posibles) para que todo funcione correctamente trabajando sobre las “nuevas” carpetas.

   Después, vuelva a dejarlo todo como estaba.
8. Cree una nueva regla llamada `test`. Cuando se ejecute

   ```
   make test
   ```

   se ejecutará el programa `Kaprekar_modular` sobre los valores 3524, 25, 1121, 6174, 111, 66565 y -33.

   Nota: Usted solo escribirá `make test` y verá cómo se ejecuta el programa y muestra el resultado.