LECCIÓN 5: Diseño de programas

María de la Paz Guerrero Lebrero

Curso 2014 / 2015

Grado en Matemáticas

maria.guerrero@uca.es

UCA
Universidad de Cádiz

Índice

- Introducción
- ¿Dónde estamos?¿Qué sabemos?
- Creación de un programa con más de un archivo
- Construcción y uso del *makefile*
- Bloques de compilación opcionales

Introducción

- Definición de diseño
- Definición de Ingeniería del diseño
- Conceptos relacionados con el diseño

Definición de diseño

- El término **diseño** admite varios significados.
  - Así, el "diseño" puede ser una actividad, la "actividad de diseñar",
  - puede ser un producto,
  - el "resultado de la actividad de diseñar",
  - o puede ser un calificativo, y en este sentido es muy común referirse a algo como "de diseño", cuando aporta una geometría, una forma o unas cualidades diferenciadoras que implican un aire de calidad y distinción.

Definición de Ingeniería del Diseño

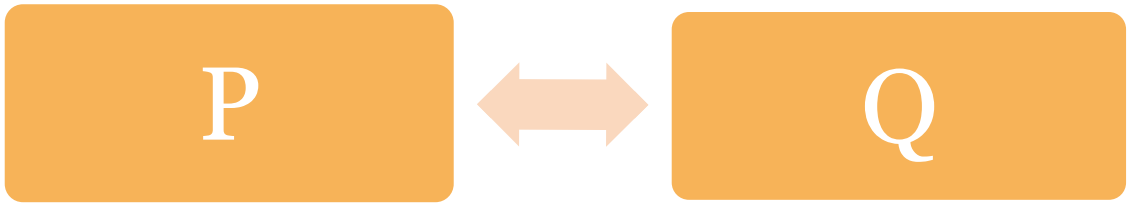
- La **ingeniería del diseño** es la representación o modelo del software, que proporciona datos sobre la estructura, arquitectura, interfaces, etc. y es utilizada por los ingenieros del software.
- Esta fase es importante ya que de aquí se extraen o establece la calidad del software y se pueden hacer las mejoras pertinentes si es necesario sin invocar a pruebas o al cliente.

Conceptos relacionados con el diseño

- **Abstracción:** Cada algoritmo es como una "caja negra". Una vez que el programa ha sido escrito es posible usarlo sin necesidad de conocer las particularidades de su algoritmo, con sólo tener una definición de la acción que realiza y una descripción de los parámetros que maneja.
- **Modularidad:** el software se divide en componentes independientes e individuales. Estos componentes llamados módulos se integran para satisfacer los requisitos del problema.

Conceptos relacionados con el diseño

- **Principio de ocultación:** El principio de ocultación de la información no sólo oculta los detalles en "cajas negras", sino que asegura que ninguna otra "caja negra" puede acceder a esos datos.

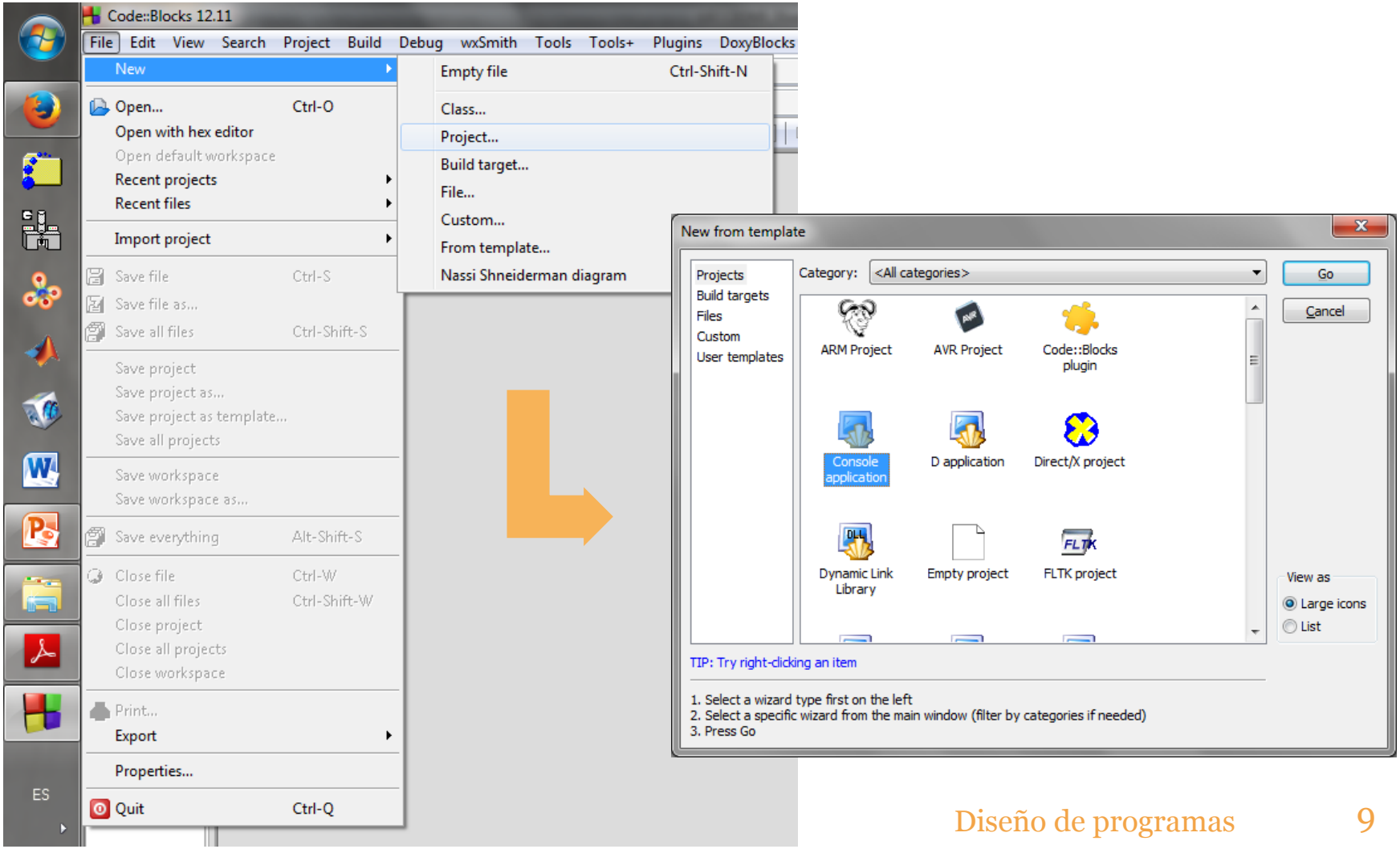


- **Q** no debe saber cómo **P** realiza la ordenación pero si que debe "pasarle" un array de enteros y de longitud MaxArray.

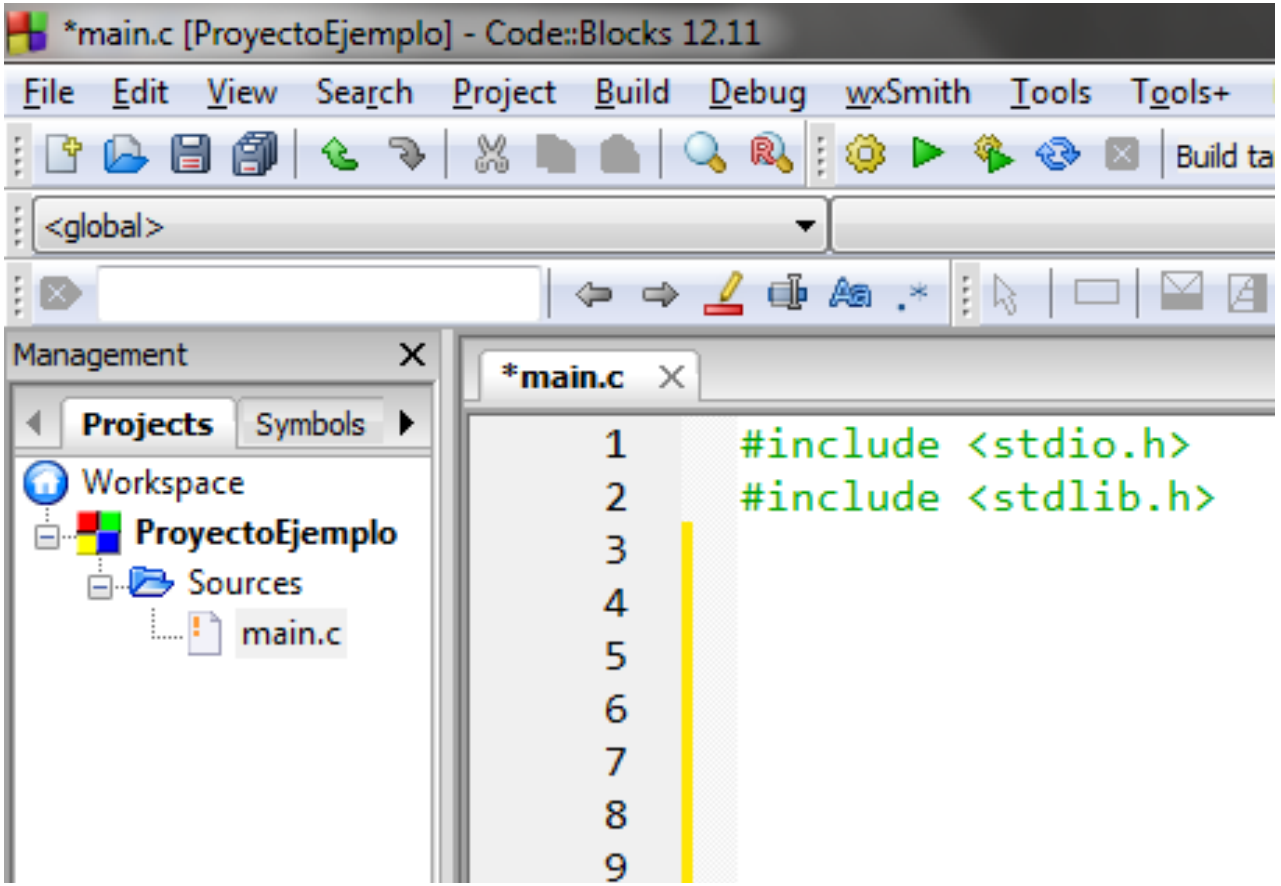
¿Dónde estamos? ¿Qué sabemos?

- Creación de un proyecto con un solo fichero
- Librerías
- Declaración y definición de funciones
- Estructura de nuestros programas

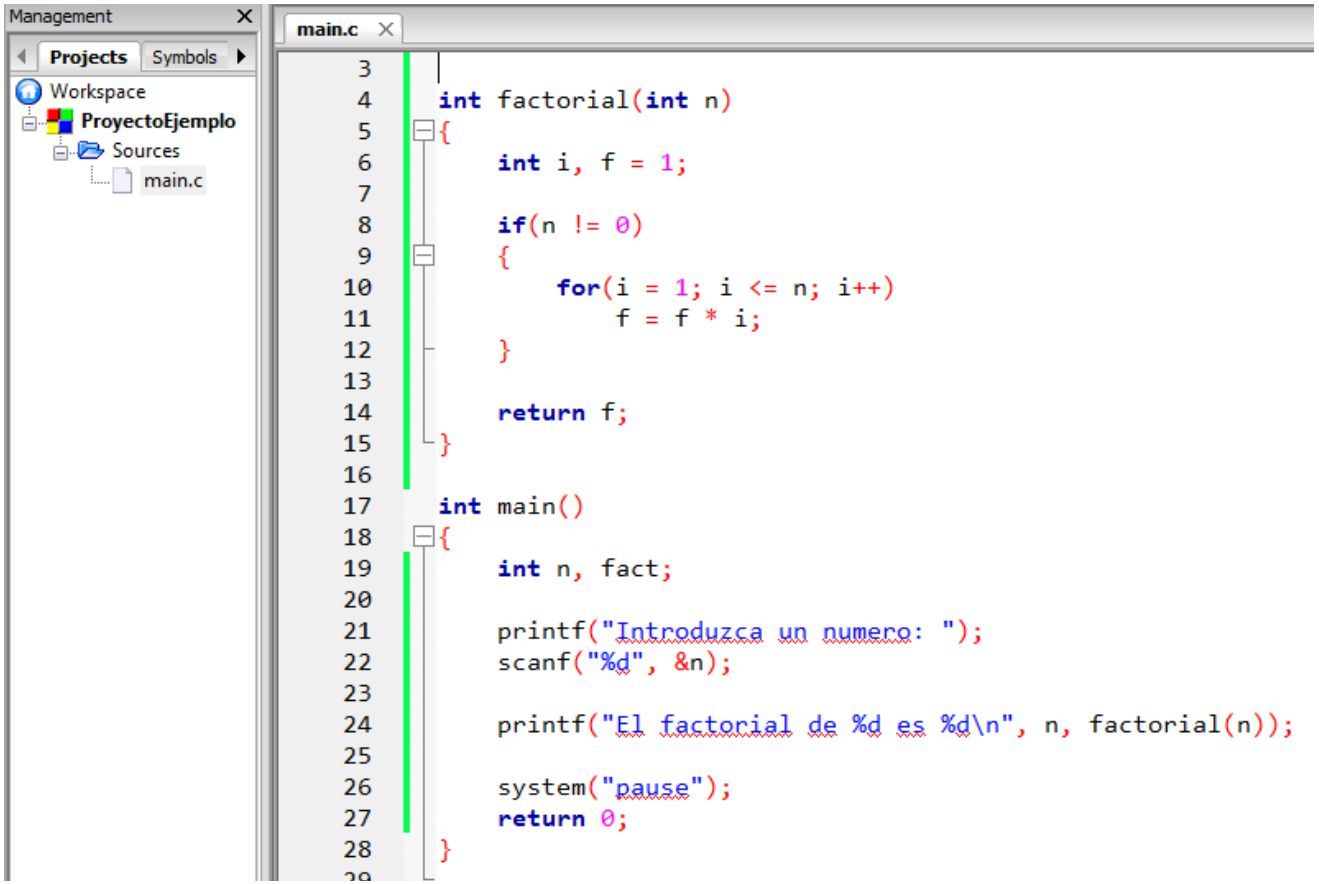
Creación de un proyecto con un solo fichero



Librerías


*main.c [ProyectoEjemplo] - Code::Blocks 12.11
File Edit View Search Project Build Debug wxSmith Tools Tools+
Build ta
<global>
Management
Projects Symbols
Workspace
ProyectoEjemplo
Sources
main.c
*main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5
6
7
8
9

Declaración y definición de funciones



Estructura de nuestros programas

1. Librerías y constantes	2. Declaración y definición de funciones	3. Funcion *main()*
• include • define	• Cabeceras • Cuerpo	• Función principal • Llamada al resto de funciones

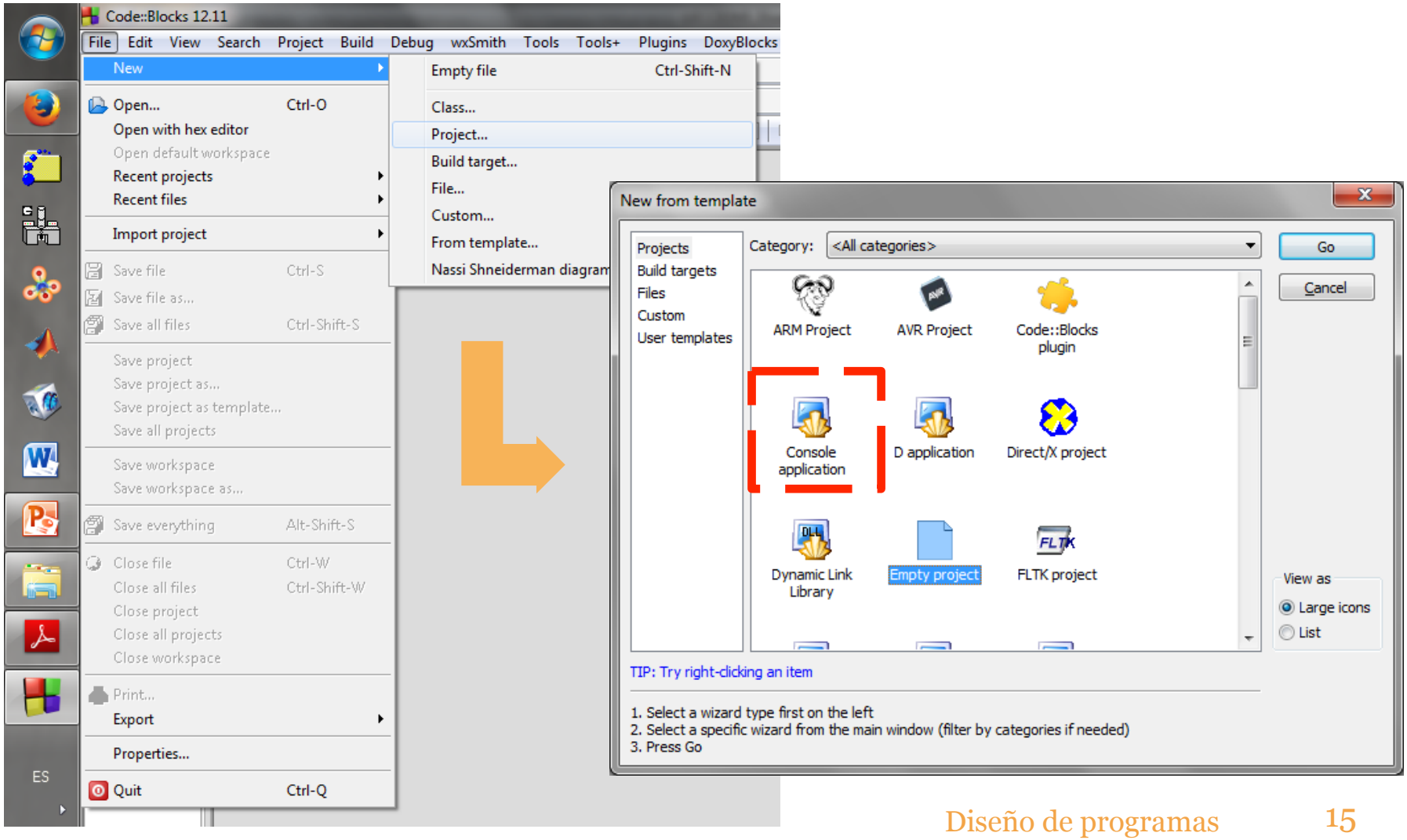
Creación de un programa con más de un archivo

- Introducción
- Empty proyect
- Fichero de cabeceras (.h)
- Fichero de funciones (.c)
- Fichero con el *main* (.c)

Introducción

- Cuando se escriben programas grandes se deberá programar en módulos.
- Estos serán archivos fuentes separados:
  - Las librerías, variables globales, tipos de datos y prototipo de funciones, deben ir en el fichero de cabeceras (.h)
  - La función *main()* deberá estar en un archivo, por ejemplo en main.c
  - Las definiciones de las funciones en otro fichero (.c)

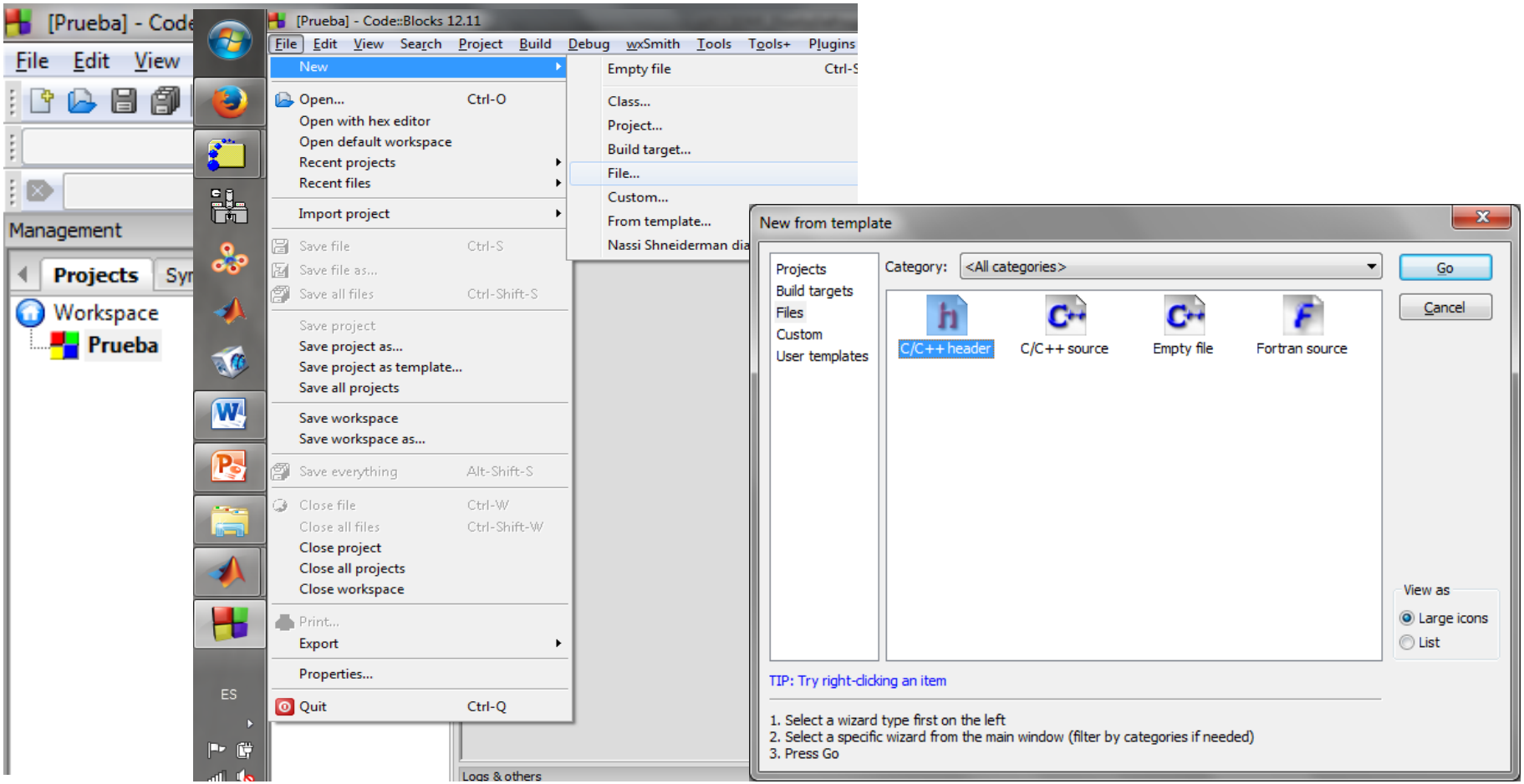
Console Application



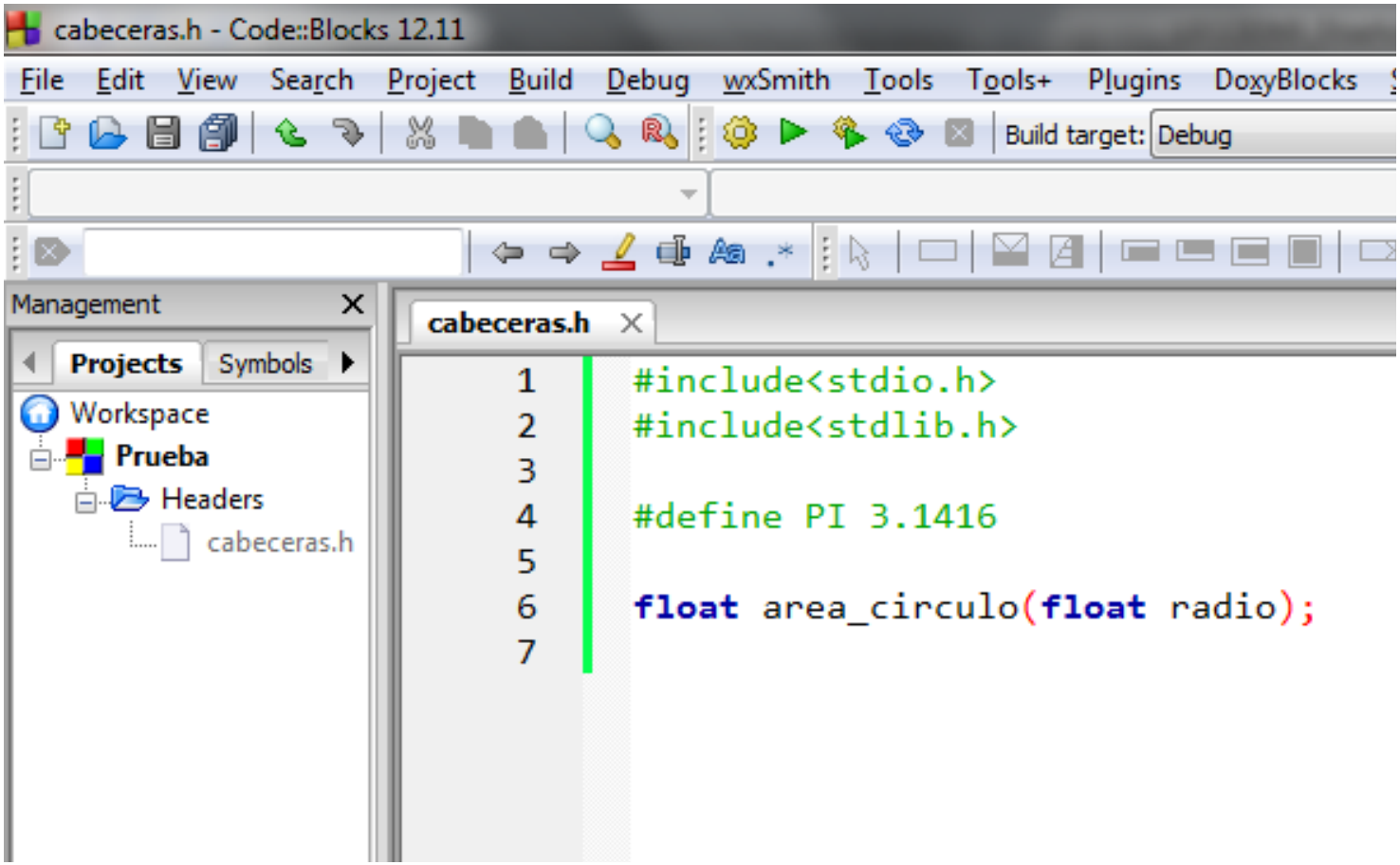
Fichero de cabeceras (.h)

- Se denomina **fichero cabecera**, especialmente en el ámbito de los lenguajes de programación C, al archivo, normalmente en forma de código fuente, que el compilador incluye de forma automática al procesar algún otro archivo fuente.
- También pueden ser creados por el programador.
- Contiene, normalmente, una declaración de las funciones a utilizar, subrutinas, variables, u otros identificadores.

Fichero de cabeceras (.h)



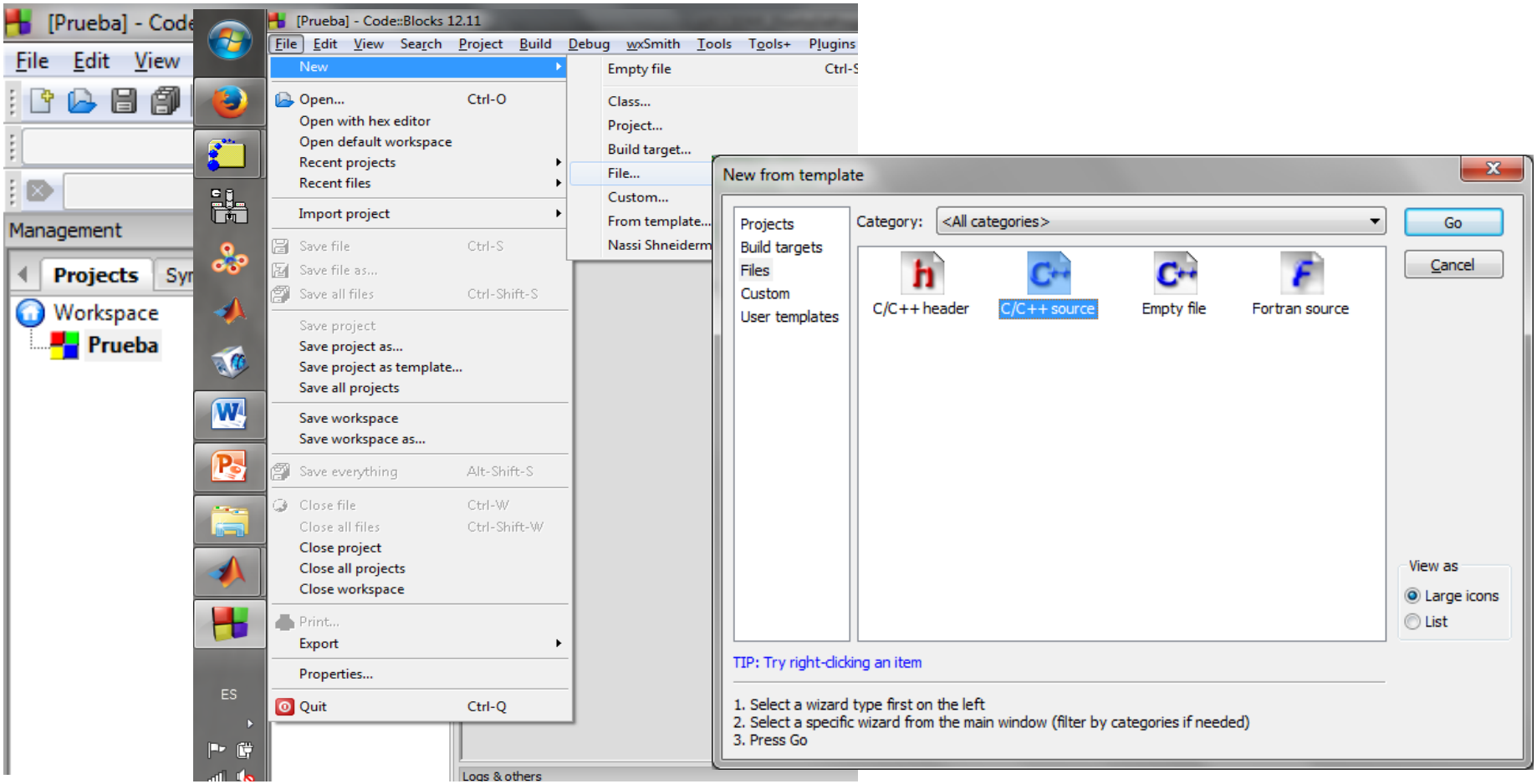
Fichero de cabeceras (.h)



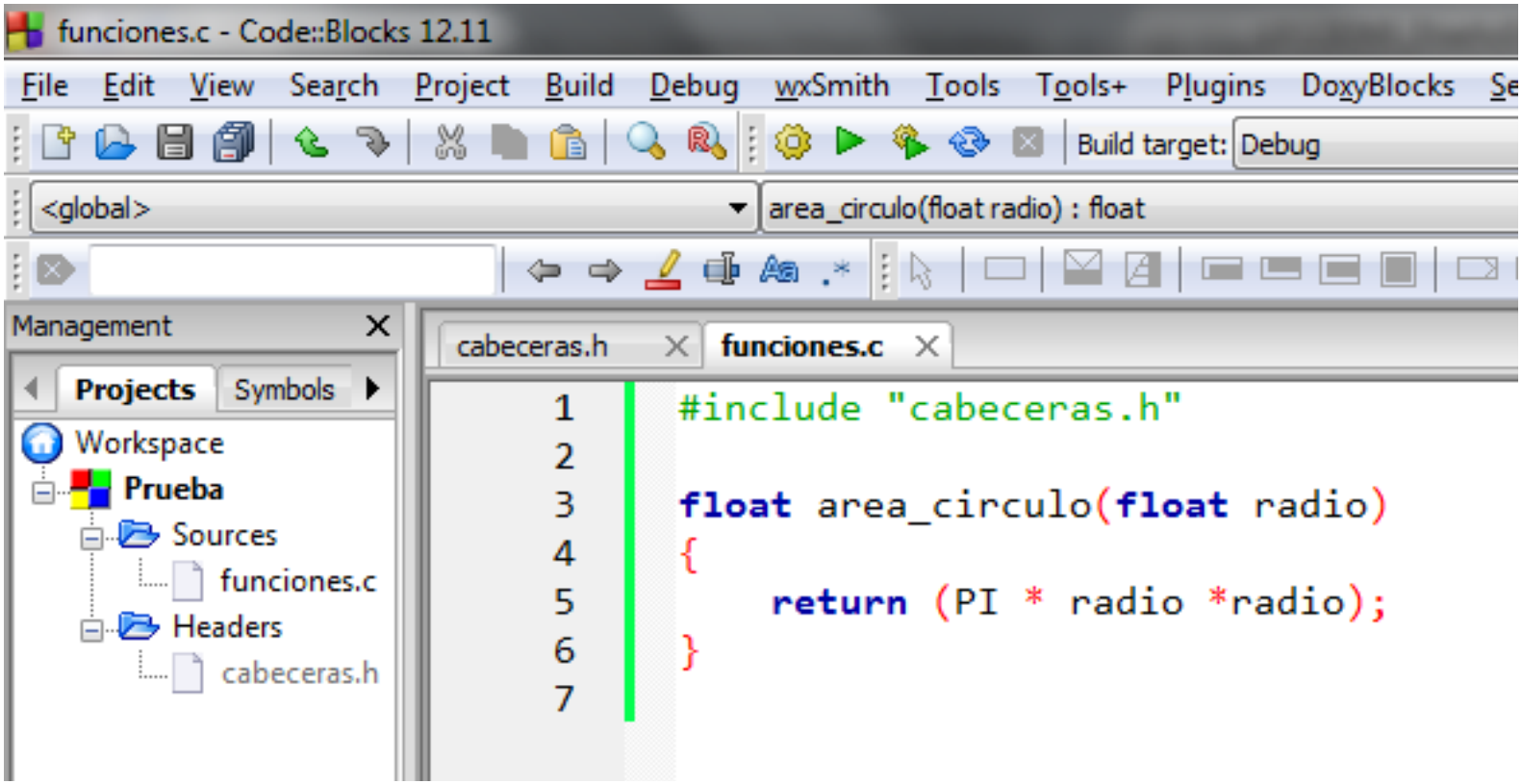
Fichero de funciones (.c)

- Contiene:
  - El fichero de cabeceras creado anteriormente (#include “cabecera.h”)
  - La definición de las funciones que se vayan a utilizar durante el programa.

Fichero de funciones (.c)



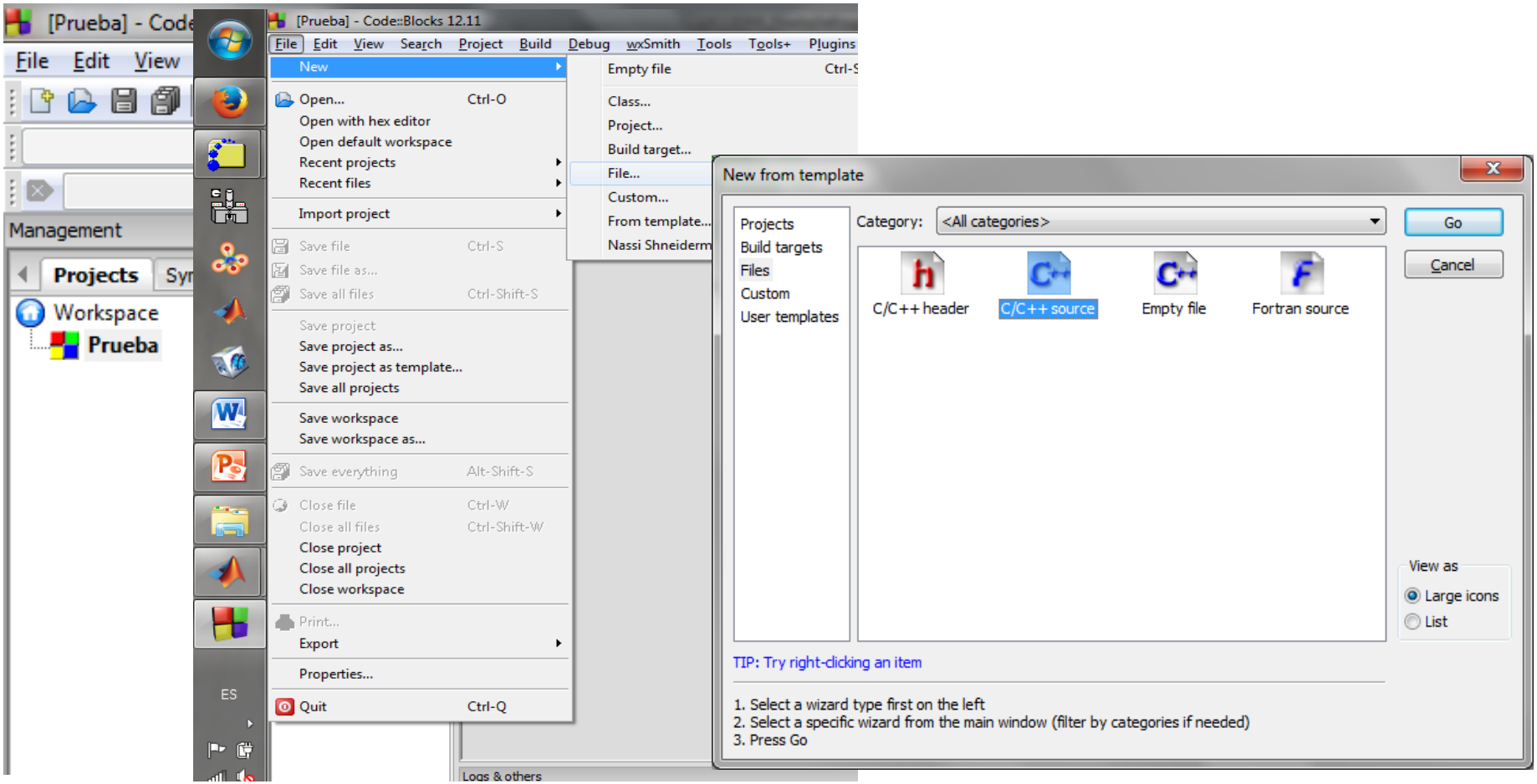
Fichero de funciones (.c)



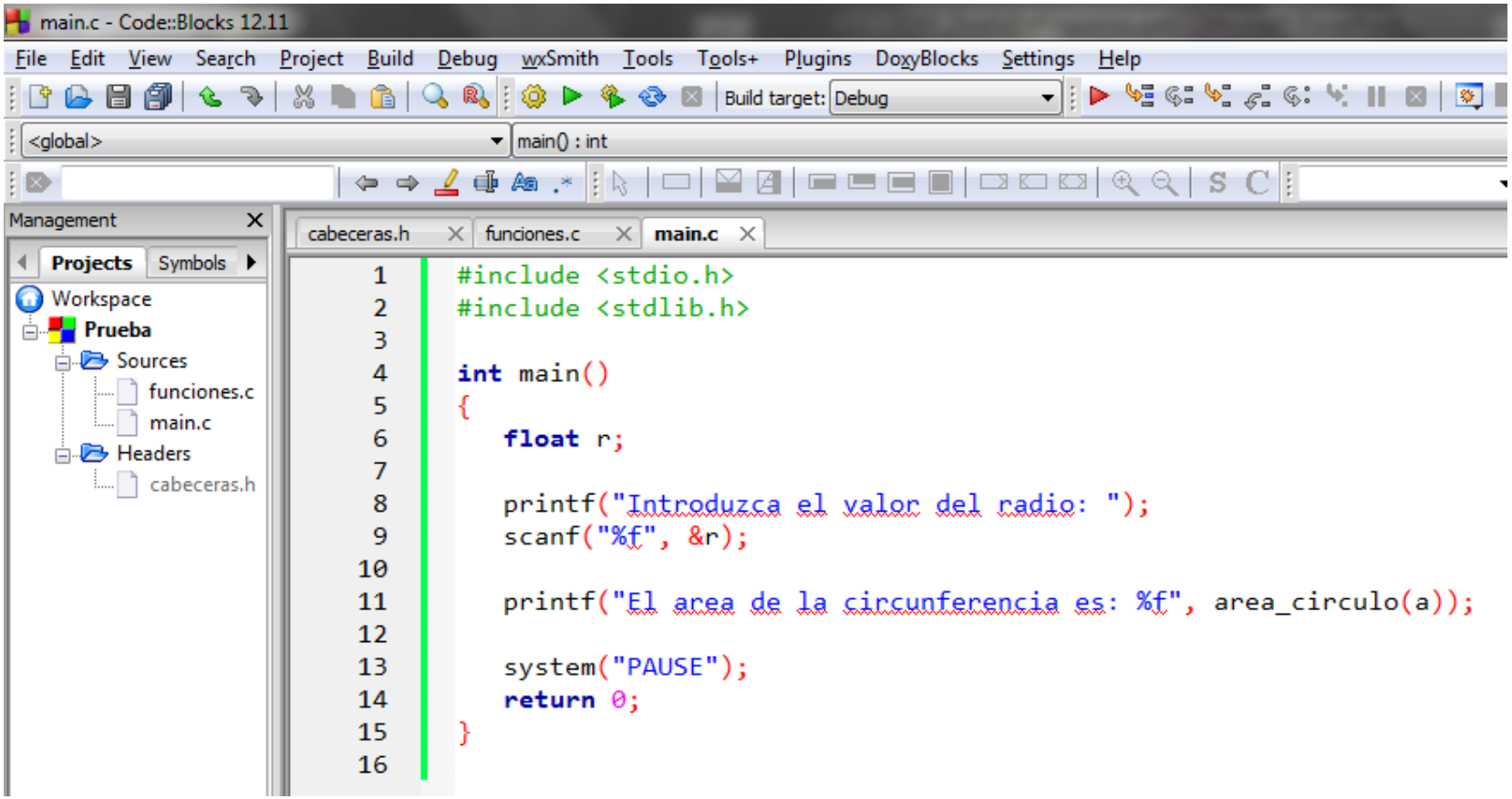
Fichero con el *main* (.c)

- Contiene:
  - El fichero de cabeceras creado anteriormente (#include “cabecera.h”)
  - La función *main()*

Fichero con el main(.c)



Fichero con el *main* (.c)



Ejercicios

- **EJERCICIO 1:** Implementa un programa que ordene un vector de números por el método de inserción.

6 5 3 1 8 7 2 4

Construcción y uso del *makefile*

- Definición de *makefile*
- Estructura de un *makefile*
- Invocando al comando *make*
- Desarrollo del *makefile*

Definición de *makefile*

- **make** es una herramienta de generación o automatización de código, muy usada en los sistemas operativos tipo Unix/Linux, aunque también puede usarse en Windows.
- Por defecto lee las instrucciones para generar un programa u otra acción del fichero *makefile*. Las instrucciones escritas en este fichero se llaman dependencias.

Definición de *makefile*

- La herramienta **make** se usa para:
  - Crear el fichero ejecutable o programa.
  - Su instalación.
  - La limpieza de los archivos temporales en la creación del fichero.
  - Etc.
- Todo ello especificando unos parámetros iniciales (que deben estar en el makefile) al ejecutarlo.

Estructura de un *makefile*

- Los *makefiles* son los ficheros de texto que utiliza **make** para llevar la gestión de la compilación de programas.
- Se podrían entender como los guiones de la película que quiere hacer **make**, o la base de datos que informa sobre las dependencias entre las diferentes partes de un proyecto.
- Todos los *makefiles* están ordenados en forma de reglas, especificando qué es lo que hay que hacer para obtener un módulo en concreto.

Estructura de un *makefile*

- El formato de cada una de esas reglas es el siguiente:

```
objetivo : dependencias
comandos
```

- En “objetivo” definimos el módulo o programa que queremos crear, después de los dos puntos y en la misma línea podemos definir qué otros módulos o programas son necesarios para conseguir el “objetivo”.
- En la línea siguiente y sucesivas indicamos los comandos necesarios para llevar esto a cabo. Es muy importante **que los comandos estén separados por un tabulador del comienzo de línea.**

Estructura de un *makefile*

- Ejemplo:

```
juego : ventana.o motor.o bd.o
        gcc –O2 –c juego.c –o juego.o
        gcc –O2 juego.o ventana.o motor.o bd.o –o juego
```

- Para crear "juego" es necesario que se hayan creado "ventana.o", "motor.o" y "bd.o" (típicamente habrá una regla para cada uno de esos ficheros objeto en ese mismo Makefile).

Estructura de un *makefile*

- **Comentarios:** Todo lo que esté escrito desde el carácter "#" hasta el final de la línea será ignorado por make.
- **Variables:** Es muy habitual que existan variables en los ficheros *makefile*, para facilitar su portabilidad a diferentes plataformas y entornos. La forma de definir una variable es muy sencilla, basta con indicar el nombre de la variable (típicamente en mayúsculas) y su valor. Ejemplo: CC = gcc –O2

Estructura de un *makefile*

- **Acceso a variables:** Para acceder al contenido de las variables se debe poner el símbolo $ seguido de ésta entre paréntesis. Ejemplo: $(CC)
- **Reglas virtuales:** No generan un fichero en concreto, sino que sirven para realizar una determinada acción dentro de nuestro proyecto. Normalmente estas reglas suelen tener un objetivo, pero ninguna dependencia.

Estructura de un *makefile*

- **Ejemplo de regla virtual:** la regla "clean" que incluyen casi la totalidad de *makefiles*, utilizada para "limpiar" de ficheros ejecutables y ficheros objeto los directorios que haga falta, con el propósito de rehacer todo la próxima vez que se llame a make

```
clean: rm –f juego *.o
```

- Esto provocaría que cuando se ejecutase "make clean", se borrase el fichero "juego" y todos los ficheros objeto.

Estructura de un *makefile*

- **Reglas implícitas:** No todos los objetivos de un *makefile* tienen por qué tener una lista de comandos asociados para poder realizarse. En ocasiones se definen reglas que sólo indican las dependencias necesarias, y es el propio make quien decide cómo se lograrán cada uno de los objetivos.

Estructura de un *makefile*

- **Ejemplo de regla implícita:**

```
juego : juego.o
juego.o : juego.c
```

- Para generar "juego" es preciso generar previamente "juego.o" y para generar "juego.o" no existen comandos que lo puedan realizar, por lo tanto, make presupone que para generar un fichero objeto basta con compilar su fuente, y para generar el ejecutable final, basta con enlazar el fichero objeto. Así pues, implícitamente ejecuta las siguientes reglas:

```
cc –c juego.c –o juego.o
cc juego.o –o juego
```

Estructura de un *makefile*

- **Reglas patrón:** Las reglas implícitas, tienen su razón de ser debido a una serie de reglas patrón que implícitamente se especifican en los *makefiles*. Se pueden redefinir esas reglas, e incluso inventar reglas patrón nuevas.

Estructura de un *makefile*

- **Ejemplo de regla patrón:**

```
%.o : %.c
        $(CC) $(CFLAGS) $< -o $@
```

- Para todo objetivo que sea un ".o" y que tenga como dependencia un ".c", ejecutaremos una llamada al compilador de C ($(CC)) con los modificadores que estén definidos en ese momento ($(CFLAGS)), compilando la primera dependencia de la regla ($<, el fichero ".c") para generar el propio objetivo ($@, el fichero ".o")

Invocando al comando *make*

- Cuando invocamos al comando **make** desde la línea de comandos ocurre lo siguiente:
  1. Se busca un fichero que se llama *GNUmakefile*, si no se encuentra se busca un fichero llamado *makefile*.
  2. Si no se encontrase, se buscaría el fichero "Makefile".
  3. Si no se encuentra en el directorio actual ninguno de esos tres ficheros, se producirá un error y **make** no continuará.

Desarrollo del *makefile*

```
# Compilador de C

CXX = gcc

# Módulos objeto y ejecutables.

OBJS = main.o funciones.o
EXES = ejecutable

# Obtención de los ejecutables.

ejecutable: main.o funciones.o
    $(CXX) -o $@ $^

# Obtención de los objetos.

$(OBJS): cabecera.h

# Limpieza del directorio.

clean:
        rm -f *.o *~ $(EXES)
```

Desarrollo del *makefile*

- **EJERCICIO 2:** Crea un fichero *makefile* con el cual se obtenga el ejecutable del programa implementado en el ejercicio 1.

Bloques de compilación opcionales

- Definición
- Tipos

Definición

- Existen sentencias de control que se incluyen en el lenguaje de macros (preprocesador) para crear bloques de compilación.

- También existen instrucciones que ayudan a llevar a cabo la compilación de los programas.

Tipos

- **Condicionales:**
  - #ifdef x– Si está definido x
  - #ifndef x – Si no está definido x
  - #else – En cualquier otro caso (si no)
  - #endif – Fin de la sentencia condicional

```
#ifdef _WIN32
      #include <windows.h>
#else
     #include <unistd.h>
#endif
```

Tipos

- **Otras:**
  - #define x valor– Definición de la constante x
  - #undef x – Eliminación de la constante x

```
#define PI 3.1416

//Uso de PI en el programa

#undef PI
```

Ejercicio

- **EJERCICIO 3:** Una matriz con m filas y n columnas tiene $m \times n$ elementos, y en C se almacena en la memoria por filas. Se pide realizar un programa que haga lo siguiente:
  - Leer los elementos de una matriz $m \times n$ *de un fichero.*
  - Leer unos valores p y q, y con los mismos elementos de la matriz $m \times n$ leída previamente, crear otra matriz de p filas y q columnas, que tenga el mismo número de elementos que la anterior (comprobar que $p \times q = m \times n$, y si no se cumple volver a leer p y q), de modo que los elementos estén almacenados en la memoria en el mismo orden, o dicho de otra forma, que el orden por filas se mantenga.

Ejercicio

El siguiente ejemplo ilustra lo que se pretende realizar. Dada la siguiente matriz *3×4*:

1 3 8 -1
6 4 2 9
6 3 -1 5

hallar una matriz *2×6* que tenga sus elementos en el mismo orden en la memoria. Solución: *3×4* es igual que *2×6*, luego los números son correctos. Los elementos de la matriz original están en la memoria ordenados por filas, es decir, su orden es:

1 3 8 -1 6 4 2 9 6 3 -1 5

Reorganizándolos como matriz *2×6* se obtiene:

1 3 8 -1 6 4
2 9 6 3 -1 5