UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2025/2026

Guion de prácticas

Interfaz Gráfica de Usuario de VSCode para el desarrollo de software en C++

Silvia Acid, Andrés Cano, Luis Castillo
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

Índice

1. Descripción

Esta sesión de prácticas está dedicada a aprender a utilizar, **Visual Studio Code**, **VSCode**, un editor de código fuente desarrollado por Microsoft. Es gratuito, de código abierto (*open source*) y multiplataforma, (funciona en Windows, macOS y Linux) [1]. VSCode es ligero y flexible, y está centrado en la **edición de código** pues, resalta sintaxis, autocompleta código y soporta muchos lenguajes. Otro atractivo es que, lleva la **depuración integrada**, esto es, permite ejecutar y depurar código directamente desde el editor y se integra con el terminal del sistema, Git y otras herramientas externas. Indicar que, puede convertirse en un entorno muy completo mediante la instalación de extensiones. Para la realización de las prácticas de la asignatura en C++ nosotros utilizaremos unas extensiones para este lenguaje.

Para concluir, se ha elegido este entorno porque, se trata de un editor ligero, personalizable, y dispone de las características mínimas para dar soporte al desarrollo de software y actualmente muy popular entre los programadores.

En este guion se explica cómo utilizarlo en un **Linux** que ya tenga instalado una serie de programas como **g++, ld, ar** [2] además del depurador **gdb**. Si aún no los tiene instalados, puede ver la sección 4 de la guía de configuración con VirtualBox y GNU/Linux proporcionado en **Prado**.

2. Objetivos

- Realizar la construcción de un primer proyecto con un único fichero fuente.
- Comprender el modelo de compilación separada en C++ y las distintas etapas (preprocesamiento, compilación, enlazado y obtención de código final).
- Aprender a modularizar y organizar el software en diversos ficheros y carpetas.
- Usar **VSCode** como editor de código fuente habitual de forma conjunta con CMake.
- Usar de forma elemental la herramienta CMake para alcanzar una mejor comprensión de las tareas que se automatizan para la obtención de código binario.
- Aprender a construir una biblioteca (library) [3].

[1] No hay que confundir **VSCode** con Visual Studio, este consiste en un IDE (*Entorno de Desarrollo Integrado*) de pago de la misma casa.

[2] Estos programas pueden encontrarse en el paquete **build-essential**.

[3] También conocido como librería, que es la traducción más común en el lenguaje coloquial para referirse a una colección de código precompilado.

3. Instalación y ejecución de Visual Studio Code

Es necesario instalar **VSCode** que puede encontrar en el sitio oficial de **Visual Studio Code** [4]. Su instalación en Linux es muy sencilla, para ello existen varias opciones dependiendo de la distribución del S.O. que se disponga. Los métodos más comunes y recomendados son usar el repositorio oficial de Microsoft (APT o DNF) o paquete `.deb/.rpm` o bien un paquete Snap. Los pasos a seguir pueden encontrarse en el mismo sitio [5].

Una vez instalado se podrá ejecutar desde el menú de aplicaciones instaladas (se recomienda una vez iniciado el programa, con el botón derecho del ratón escoger 'Añadir a los favoritos' para añadir un acceso rápido en la barra de herramientas de Linux). También podrá ejecutarse desde la línea de comandos con:

```
/usr/bin/code
```

o simplemente

```
code
```

VSCode en este estado es poco útil, por lo que para facilitar el desarrollo y mantener un código en C++ claro y estandarizado, es necesario instalar las siguientes extensiones:

Figura 1: Icono de extensiones

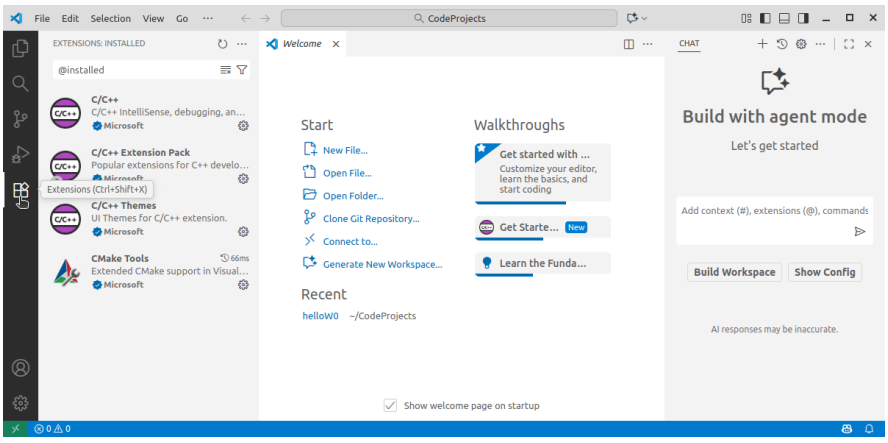

Figura 2: Interfaz **VSCode** con las extensiones necesarias

Para ello, basta con seleccionar Extensiones [6] en el panel de Actividad-

[4] Enlace https://code.visualstudio.com/
[5] Enlace https://code.visualstudio.com/docs/setup/linux
[6] Extensiones

des, buscar la extensión C/C++ Extension Pack32 (Microsoft) que contiene varias extensiones a su vez (C/C++, C/C++ themes , CMake , CMake tools) o separadamente como se muestra en la imagen 2 y seleccionar `Install` (instalar) [7] .

4. Explorando la GUI de Visual Studio Code

En la Figura 3 podemos seguir el ciclo de edición, compilación y posterior ejecución para un programa muy simple, en una instalación de Linux estándar. Se habla de *cadena de herramientas* (toolchain) al conjunto de programas y utilidades que se utilizan en orden para compilar, enlazar y obtener un producto de software. Aunque la compilación manual con el

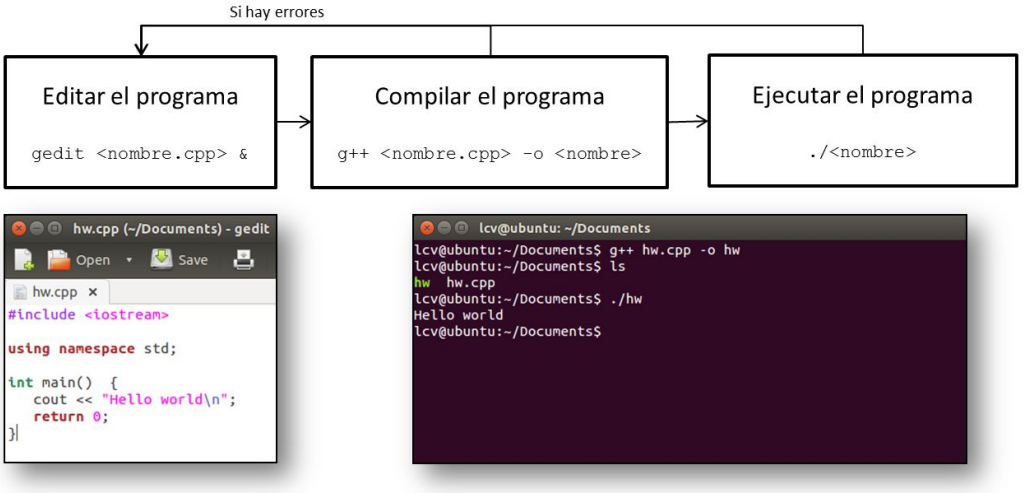

Figura 3: Cadena de herramientas incluye la compilación, y ejecución de un programa simple en C++ en Linux

comando g++ en la línea de comandos ofrece un control total y es fundamental para comprender el proceso, optaremos por utilizar la Interfaz Gráfica de Usuario, (**GUI**), de **VSCode** para realizar de forma sencilla y **automática** las operaciones de **compilación, enlazado, ejecución** y como veremos más adelante, **depuración** [8].

Pero antes, por una cuestión de organización y mantenimiento del código a largo plazo, vamos a disponer de una carpeta dedicada exclusivamente a los proyectos de desarrollo de C++, nos referiremos a ella como *CodeProjects*.

[7] Las extensiones indicadas, son todas de autoría probada luego, pueden instalarse sin restricciones.

[8] Asumiendo que ya están operativos: VSCode, las extensiones indicadas y ya disponemos de las herramientas g++ y gdb...

4.1. La primera ejecución

Desde la interfaz, **Open Folder**, abrimos la carpeta *HelloW* [9], que se encuentra dentro de *CodeProjects* , y en el editor podemos ver el contenido del archivo *helloWorld.cpp* como se muestra en la Figura 5.

Figura 4: Icono de Archivos

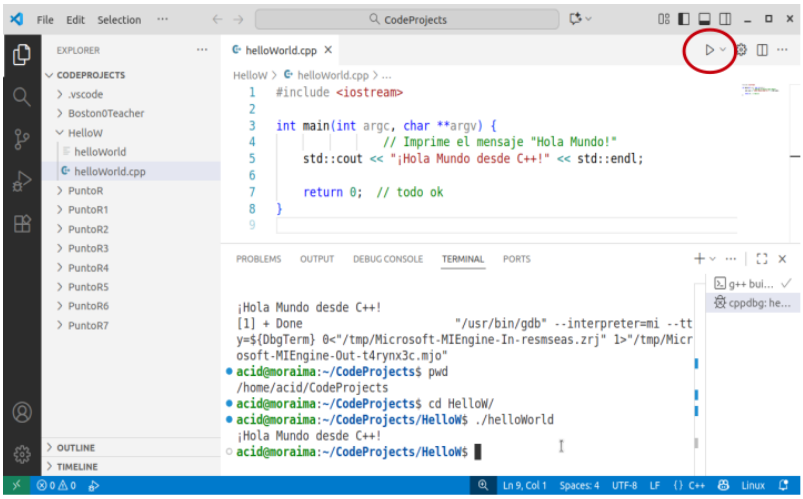

Figura 5: Ventana de edición. Edición del archivo helloWorld.cpp y procesamiento automático del código fuente para su ejecución.

Con un programa tan simple, ya se puede proceder a su ejecución, sin más. Existen muchas formas de hacerlo pero, la más simple es pinchando en el icono señalado en la Figura 5, `Run C/C++ File`. Con ello, se han realizado de forma automática los procesos indicados en la figura 3, y el binario obtenido coincide con el nombre del fuente (sin extensión), *helloWorld* [10]. Tras su ejecución se obtiene la salida del programa en un terminal, TERMINAL de la Figura 5. Se trata de una línea de comandos situada inicialmente en la carpeta *HelloW*, y desde ella se puede ejecutar el mismo programa, invocando el ejecutable, véanse detalles en el terminal de la misma figura.

Dada la cantidad de áreas de interés en la interfaz de usuario (*GUI*) de **VSCode**, vamos a describir la distribución principal de las áreas de trabajo que hay en ella. Para ello, vamos a usar como referencia la Figura 6 donde se han señalado en rojo e identificado con números cada área.

Distribución principal de las áreas de trabajo en **VSCode**:

1. **Barra de Título**. Muestra el nombre del archivo y del proyecto actual.
2. **Barra de Menú**. Da acceso a todas las funciones principales del editor: File, Edit, Selection, View, Go, Run, Terminal y Help, que co-

[9] Indicar que esta y la siguiente van a ser las únicas que no son proyectos dentro de *CodeProjects*.

[10] Puede comprobar la aparición del nuevo *helloWorld* en el panel lateral.

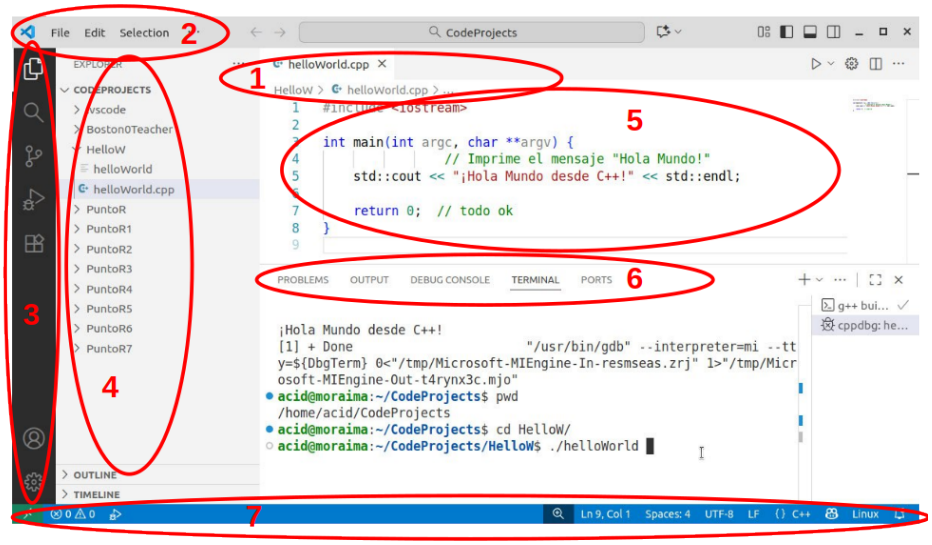

Figura 6: Distribución de las áreas de trabajo en **VSCode**

responden respectivamente a Archivo, Editar, Selección, Ver, Ir, Ejecutar, Terminal y Ayuda en castellano.

3. **Panel de Actividades**. Permite cambiar entre las vistas clave: `Explorer`, Search, Source Control (Git), Run and Debug, Extensions... y CMake, representados por iconos.

4. **Panel Lateral**. Contiene la lista de proyectos abiertos en el mismo espacio de trabajo, destacando el que se está editando.

5. **Ventana del Editor**. Es la única área que se puede dividir en varios grupos, para trabajar con múltiples archivos simultáneamente.

6. **Panel Inferior**. Contiene herramientas secundarias pero esenciales: Terminal (para comandos y pantalla de salida), Problems (errores/-warnings), Output (logs de compilación/ejecución) y Debug Console.

7. **Barra de Estado Inferior**. Información y configuración rápida. Muestra el lenguaje de programación del archivo actual, la codificación, el número de línea/columna, el estado de Git y opciones para configurar rápidamente la indentación o el servidor.

4.2. Un programa no tan simple

Dado el contenido de **MPGeometry0** que se muestra en la Figura 7, seleccionamos `Open Folder`, para abrir el directorio y editar el fichero `main.cpp` seleccionando este desde el panel lateral.

El objetivo del programa, es calcular la intersección de dos rectángulos, leer una serie de puntos y calcular cuántos de estos caen dentro de la intersección. Para ello, se usan las clases **Point2D** y **Rectangle** que aparecen en el código escrito en un único fichero mostrado en la sección 8.

```
MPGeometry0/
├── documentation.doxy
├── main.cpp
├── v_0inside.test
├── v_1inside.test
├── v_4inside.test
└── v_empty.test
```

Figura 7: a) Contenido original del **MPGeometry0** que contiene un único fuente *main.cpp*, varios ficheros de datos para evaluar el programa `*.tests` y un fichero de configuración de Doxygen, `*.doxy`.

Con objeto de explorar la GUI, y profundizar en los conocimientos de la asignatura, se recomienda realizar las siguientes tareas:

1. Comprender el contenido de *main.cpp*
   - Identificar las clases que contiene y sus métodos. Métodos **const** y no *const*.
   - Identificar las funciones externas.
   - Comprender las acciones de la función `main`, qué datos lee y cómo, el listado completo se encuentra en la sección 8.
2. Abrir el fichero *v_0inside.test*.
3. Ejecutar el main de MPGeometry0, con los datos de *v_0inside.test*. Con `Run C/C++ File` e introduciendo los datos de entrada de *v_0inside.test*. Abrir para ello el fichero indicado en el editor y copiar en el terminal las coordenadas de entrada.

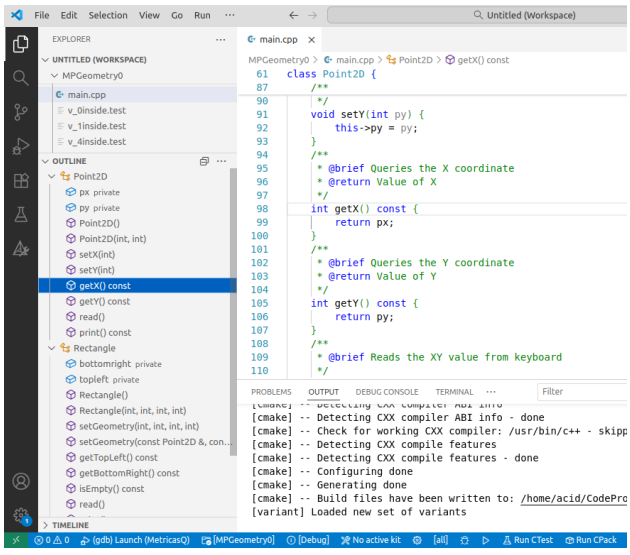

Figura 8: Lista de declaraciones del fichero abierto: clases, structs, funciones, métodos, variables, campos etc. Cada entrada tiene un icono que indica su tipo (método, campo, función...), el orden y la presencia de tipos puede cambiar.

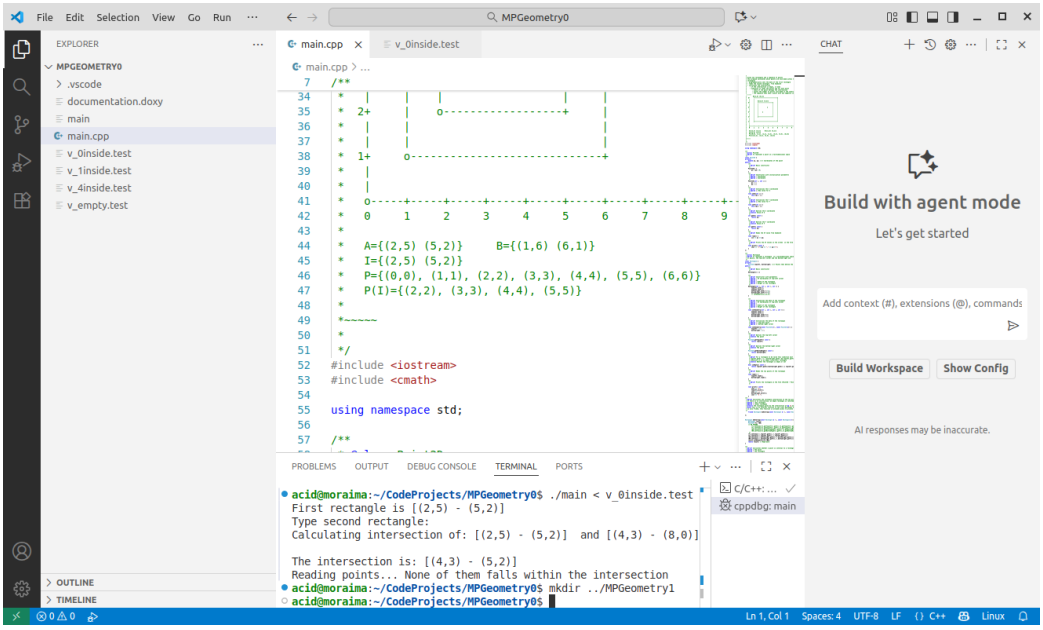

Figura 9: Cadena de herramientas con las opciones por defecto y entrada de datos por teclado. Nueva ejecución desde el terminal, con redireccionamiento de entrada de datos desde un fichero.

Para ayudar a la identificación de clases y métodos se recomienda desplegar la pestaña **OUTLINE**. En la parte izquierda aparece la lista de símbolos (declaraciones) del fichero abierto: clases, structs, funciones, métodos, variables, campos... ver Figura 8. Si se pincha con el ratón en un símbolo de la lista, se muestra en el editor tal símbolo, lo que nos permite explorar selectivamente los métodos según nuestro propio criterio.
Nota: Se recomienda vivamente tener **Outline** desplegado para navegar por el código y entre ficheros, como se verá más adelante, se ahorrarán así, kilómetros de scroll.

En la Figura 9 podemos ver una nueva forma de ejecutar el programa desde el terminal leyendo datos desde un fichero con redireccionamiento, $<$.
En la elaboración automática del ejecutable a partir del fuente (cadena de herramientas), las opciones hasta ahora son por defecto. No hay elección de cómo se denomina el ejecutable, ni dónde se sitúa, para invocarlo desde la línea de comandos.

4.3. Creación de un proyecto

Como preparación para el crecimiento de proyectos más complejos (escalabilidad), así como para mejorar la portabilidad y la simplificación del flujo de trabajo, crearemos nuestro **primer proyecto**. Este contendrá la misma colección de ficheros de `MPGeometry0`, pero con una estructura más organizada. Para su gestión personalizada, utilizaremos **CMake**, una de las extensiones que hemos instalado y a la que debemos prestar cierta atención.
A continuación, se detallan los pasos para configurar el proyecto:

1. Comenzamos **Creando el directorio, MPGeometry1**, como subdirectorio de *CodeProjects*, desde un terminal [11].

2. **Abrimos el directorio** como un nuevo espacio de trabajo. En la barra de menú, se selecciona `File`, `Open Folder` y seleccionamos desde el navegador de archivos nuestro flamante directorio **MPGeometry1**, completamente vacío. Véase Figura 10.

3. Solicitamos apoyo para la configuración mínima necesaria para CMake, el asistente **Quick Start**. En la barra de menú, se selecciona `View`, `Command Palette`, `Quick Start` y seleccionamos una a una las opciones que se detallan a continuación. Véase Figura 11.

   - Introducción de `MPGeometry1` como **nombre del proyecto**. En este caso, van a coincidir el nombre del proyecto con el nombre del directorio [12].
   - Indicar que se trata de un proyecto C/C++.
   - Indicar que se trata de la construcción de un ejecutable.
   - Seleccionamos opciones adicionales de Tests y CPack, como última entrada [13].

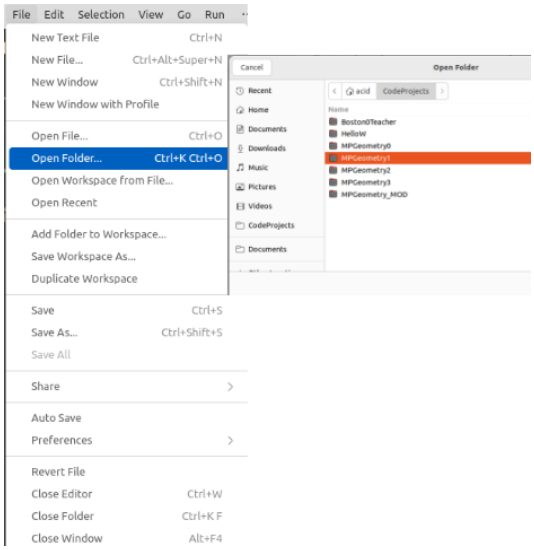

Figura 10: Se establece el directorio de referencia, una vez creado el directorio.

Una vez completados los pasos, el directorio `MPGeometry1` , que inicialmente estaba vacío, ahora contiene un subdirectorio (`build`) y dos

[11] Véase, el comando de la última línea del terminal en la figura 8.

[12] El nombre del proyecto y el nombre del directorio podrían ser diferentes, en IDE como Netbeans tienen vistas‘ separadas. El nombre del proyecto puede determinar el nombre del ejecutable, es el nombre lógico, mientras que el nombre del directorio, un contenedor, no es más que la parte física del proyecto.

[13] No se darán detalles al respecto, se deja al estudiante investigar su utilidad.

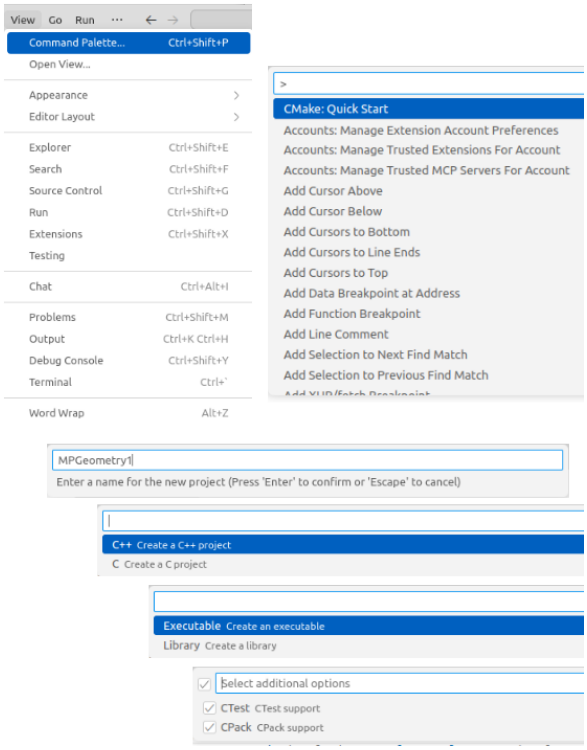

Figura 11: Se designa el proyecto y se configura de forma mínima para que CMake pueda intervenir.

nuevos ficheros (`CMakeLists.txt` y `main.cpp`), cuyos contenidos se muestran en la Figura 12.

VSCode integra **CMake**, ofreciendo una forma automatizada y sencilla de gestionar proyectos. Es importante aclarar que **CMake** no compila directamente el código, en su lugar, lee los archivos de configuración `CMakeLists.txt` y genera los archivos necesarios para que el entorno de construcción nativo (como Make, Ninja, etc.) pueda realizar la compilación, el enlazado (link) y la producción del software.

En este momento disponemos de un **CMakeLists.txt** operativo, sin errores, y que podemos editar y modificar a nuestro antojo con la sintaxis de comandos de CMake [14].

Contenido de `CMakeLists.txt`:

```
# Versión mínima de CMake requerida
cmake_minimum_required(VERSION 3.10)

# Define el nombre del proyecto
project(MPGeometry1 VERSION 0.1.0 LANGUAGES C CXX)

# Agrega el ejecutable MPGeometry1, podía ser add_library() si no seleccionamos ejecutable
add_executable(MPGeometry1 main.cpp)

# Otras configuraciones básicas
# para tests y para empaquetar software y para su portabilidad
```

[14] Para una ampliación de comandos, consultar manuales oficiales de **cmake.org**, https://cmake.org/cmake/help/latest/manual/cmake-commands.7.html.

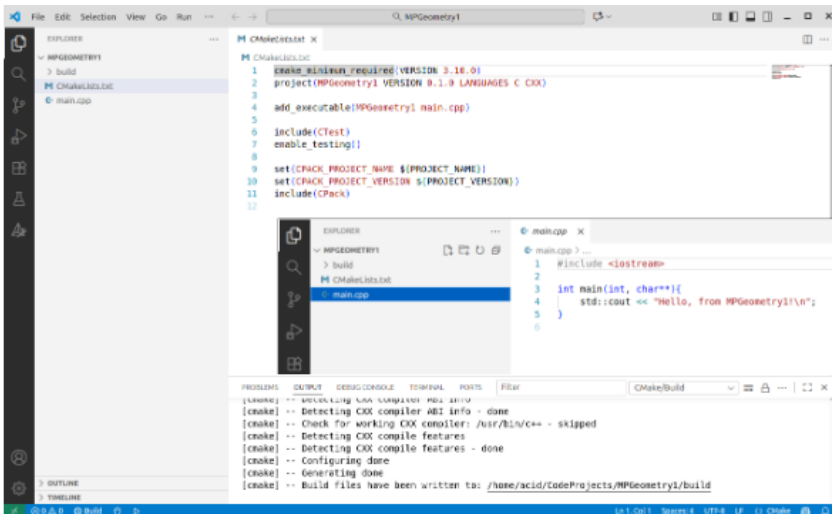

Figura 12: Contenido actual del directorio `MPGeometry1`: nuevo directorio `build`, texto de `CMakeLists.txt`, y del fuente `main.cpp`.

Como podemos observar **CMakeLists.txt** contiene unas directivas esenciales para la herramienta **CMake**, se trata de comandos que especifican la versión mínima requerida de **cmake** para su operatividad (línea 2), el nombre del proyecto (línea 5), qué archivo fuente compilar: `main.cpp` y qué binario generar: `MPGeometry1` (línea 8).

El fuente **main.cpp**, cortesía también del asistente `Quick Start` consiste en una nueva versión del `helloWorld.cpp` adaptado al proyecto `MPGeometry1` que se acaba de crear.

Otra de las operaciones que se ha llevado a cabo de forma automática es la generación de un ejecutable, tenemos eco de ello en el panel inferior, ventana `OUTPUT`, visible en la Figura 12.

Con la ejecución de CMake: Configure, se crea el directorio de construcción (por defecto `build` en la raíz del proyecto) y en él se generan los archivos del sistema de compilación nativo (como Makefiles o archivos de Ninja) además del ejecutable. Puede comprobar su existencia desde el terminal mediante el comando

```
ls MPGeometry1
```

Pero este no es el contenido que deseamos para `MPGeometry1`, sino el que se detalla en el árbol **a)** de la Figura 13. Por ello vamos a tomar como referencia el contenido original del directorio `MPGeometry0` árbol **b)** y a recrear la estructura de directorios y su contenido con ayuda de la GUI.

Podemos incorporar el directorio `MPGeometry0` en nuestro espacio de trabajo actual (WorkSpace) y así disponer de los dos directorios simultáneamente y manejar de esta forma el panel lateral como un navegador de archivos. Seleccionamos en la barra de Menú: {`File`}, {`Add Folder to WorkSpace...`} se selecciona `MPGeometry0`. A continuación, cree la estructura de directorios mostrada la Figura 13 **a)**. Respete la posición relativa de cada objeto en la estructura, al copiar los archivos

```
MPGeometry1/
├── doc
│   └── documentation.doxy
├── src
│   └── main.cpp
└── tests
    ├── v_0inside.test
    ├── v_1inside.test
    ├── v_4inside.test
    └── v_empty.test
```

a)

```
MPGeometry0/
├── documentation.doxy
├── main.cpp
├── v_0inside.test
├── v_1inside.test
├── v_4inside.test
└── v_empty.test
```

b)

Figura 13: **a)** Estructura requerida de carpetas para la gestión de un proyecto que incluyen las carpetas (**doc**, **src** y **tests**); **doc** (para almacenar ficheros Doxygen y documentación del proyecto), **src** (para tener localizado el fuente) y **tests** (ficheros que contienen tests para comprobar el funcionamiento del ejecutable). **b)** Contenido original del **MPGeometry0**.

en sus carpetas respectivas.

Nota1: Debe eliminar el `main.cpp` de **MPGeometry1** versión de `hello-World`.
Nota2: Sitúe la copia del `main.cpp` de **MPGeometry0** dentro de `src`.

Con la estructura de ficheros del árbol **a)** en el proyecto, la GUI nos advierte desde distintas áreas de que, hay algún error en la construcción del ejecutable, véase la Figura 14 con zonas remarcadas en rojo.

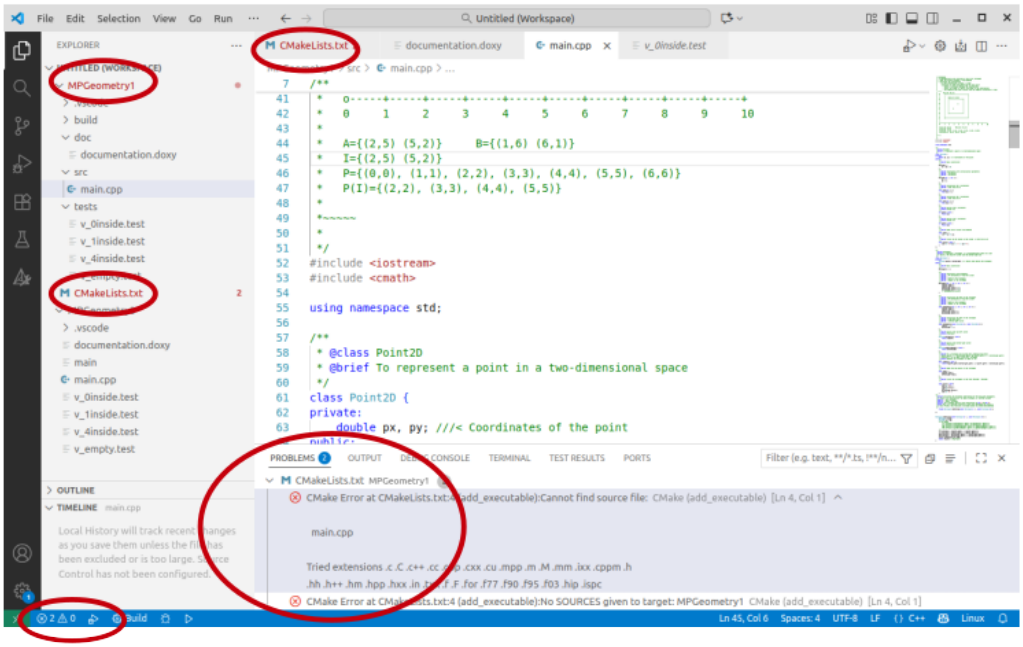

Figura 14: Problemas en el proyecto MPGeometry1, 2 errores con cmake, aviso en el fichero **CMakeLists.txt**

Figura 15: Icono de CMake

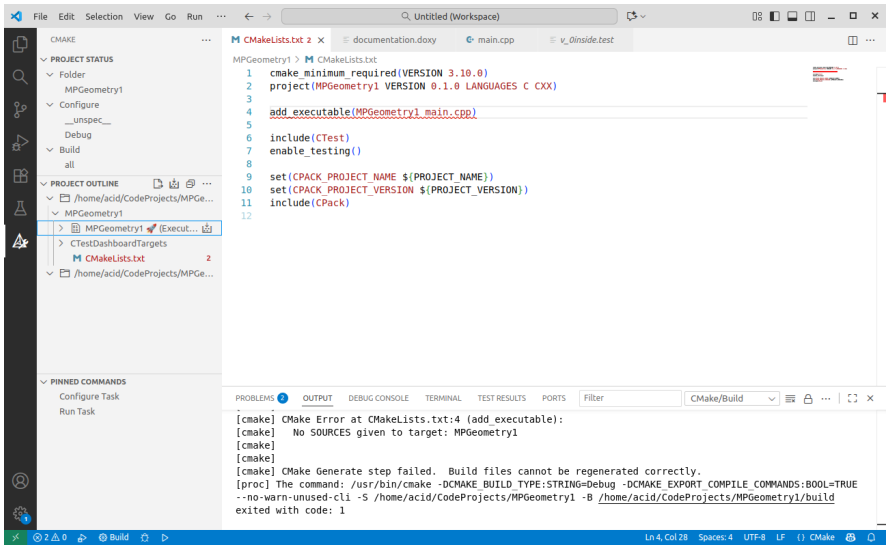

Figura 16: Edición de **CMakeLists.txt** y objetivos de CMake en el panel lateral.

El origen del error está en el fichero **CMakeLists.txt**, en la línea 4 del editor. El problema es que CMake no encuentra el fichero fuente `main.cpp` en el directorio raíz, dónde se especificaba que estaba. Es necesario corregir el comando `add_executable()`.

En la Figura 17 se muestra un extracto de las líneas más relevantes de `CMakeLists.txt` incluida la corrección y la incorporación de nuevas especificaciones para el compilador (línea 8 y línea 14). De ahora en adelante siempre utilizaremos estos dos comandos `set()` que afectan al compilador y nos facilitarán la escritura correcta y la detección de errores en nuestro código.

```
1  # Versión mínima de CMake requerida
2  cmake_minimum_required(VERSION 3.10)
3
4  # Define el nombre del proyecto
5  project(MPGeometry1 VERSION 0.1.0 LANGUAGES C CXX)
6
7  # uso de C++ 17
8  set(CMAKE_CXX_STANDARD 17)
9
10 # uso de directivas de compilación
11 # -Wall aviso de warnings,
12 # Wextra aviso de errores más sutiles
13 # -Wpedantic uso de ansi C++
14 set(CMAKE_CXX_FLAGS "-Wall -Wextra -Wpedantic")
15
16 # Agrega el ejecutable MPGeometry1
17 # Toma siempre la raíz como punto de referencia
18 add_executable(MPGeometry1 src/main.cpp)
```

Figura 17: Edición parcial de `CMakeLists.txt` con la corrección y nuevas especificaciones.

Para terminar, es necesario guardar las modificaciones, mediante la opción de menú {`File`}, {`Save`}. CMake detecta los cambios y reconstruye automáticamente el ejecutable, como puede verse en la ventana OUTPUT de la figura 17.

4.4. Construyendo y ejecutando el binario

De ahora en adelante, todas las prácticas tendrán un proyecto, y por tanto dispondrán de un fichero **CMakeLists.txt** necesario para **CMake**. Los botones de la **barra de estado** son nativos de la extensión `CMake Tools` y se activan al configurar un proyecto. Por tanto, hemos de usar estos botones de la barra de estado para la construcción y ejecución como alternativa a Code Runner de `C/C++` [15] que viene en la GUI por defecto.

Una vez configurado, se puede seleccionar el proyecto con el que operar desde el panel lateral de CMake o bien a través de los iconos de la barra de estado, como puede verse en la Figura 18.

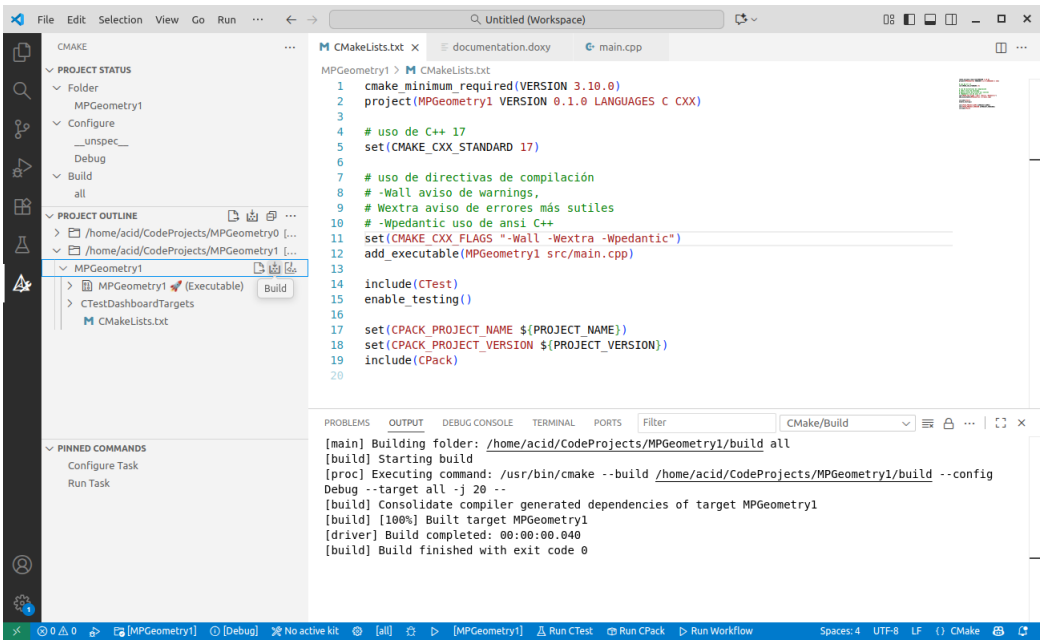

Figura 18: Varios objetivos de **CMake**: `build, clean`.

Los comandos de CMake a los que sustituye alguno de los iconos disponibles son:

```
1 ~/CodeProjects/MPGeometry1$ cmake -S . -B build
2 cmake --build build/
```

Línea 1) indica a CMake con `-S` dónde encontrar el archivo de configuración principal, `CMakeLists.txt`, `.`, con `B` que cree el `build` y que deje allí todos los archivos generados aislados del resto.
Línea 2) instrucción que realmente compila el proyecto.

```
cmake --build . --target clean
```

[15] Botón de ejecución en la esquina superior derecha, que ya no se comporta correctamente...

Ejecuta el objetivo clean (existe por defecto), cuyo propósito es dejar el directorio de construcción limpio de artefactos generados, el ejecutable entre otros...

Una vez obtenido el ejecutable, se puede ejecutar el programa desde el terminal mediante el comando:

```
~/CodeProjects/MPGeometry1/build$ ./MPGeometry1 < ../tests/v_0inside.test
```

esto es, utilizando un fichero de entrada como `v_0inside.test` como entrada. Es necesario utilizar el path ya que nuestro ejecutable, **MPGeometry1**, recién generado se encuentra en la carpeta `build` y es ahí dónde se sitúa el terminal.

Otra forma alternativa de ejecución, es mediante la selección con ratón de algún `launch` (inicio) representado mediante un cohete, pero este modo requiere de datos de entrada por teclado, explore el panel lateral...

5. Un proyecto de compilación separada

5.1. Buenos hábitos, la compilación separada

Observemos el `main.cpp` de **MPGeometry1** de aproximadamente 300 líneas, que utiliza las clases **Point2D** y **Rectangle**, con una función `main()` todo ello incluido en en un único fichero fuente. De hecho, este código así planteado representaría una mala praxis en programación [16].

Lo habitual, en proyectos medianos o grandes, es encontrarnos la implementación separada en distintos módulos o archivos, donde la norma es incluir en un único módulo todos los elementos que pudiesen estar relacionados. Pero ¿Qué es un módulo? Normalmente un módulo se divide en dos partes bien diferenciadas:

- Un archivo cabecera (con extensión .h): contiene la declaración de la clase o funciones, junto con la documentación de las mismas, esto es la sintaxis (cómo se referenciarán) y la semántica (cómo se interpretarán). Pero NO se dan detalles de cómo las operaciones deben ser implementadas.
- Un archivo de implementación (con extensión .cpp): que contiene la implementación de cada uno de los métodos de la clase o de las funciones que han sido declaradas en el fichero cabecera.

Así, un módulo puede contener una clase y algunos métodos relacionados con ella, como por ejemplo el módulo `string, vector,...`, o solo un conjunto de funciones, como por ejemplo `cmath` o `utility`.

Aunque podríamos tener libertad para asignar el nombre para los distintos archivos, es conveniente utilizar la misma etiqueta y que esta nos ayude a identificar el propósito del módulo.

[16] Va en detrimento de la escalabilidad, la manejabilidad y la colaboración en el desarrollo del software.

Además, para completar la aplicación necesitaremos al menos de otro fichero adicional donde se implementa la lógica de la misma (programa principal) y que contenga la función `main`. Este programa principal, que se escribe en otro archivo separado, se puede llamar como queramos pero normalmente se denomina `main.cpp` para hacer explícito que allí encontraremos la función main.

5.1.1. Ocultamiento de Información

Cuando separamos la especificación de la implementación (.h y .cpp) estamos separando el cómo se utiliza una función/clase del cómo se implementa. A esto se le conoce como **ocultamiento de información**, que es uno de los principios que rigen la Metodología de la Programación. Permite hablar de dos vistas, la vista 'externa' que sería la que conocen todos los usuarios del módulo (.h), indicando exactamente qué se puede utilizar y cómo (por eso la documentación se incluye en este fichero) y la vista 'interna', que es la que conocen únicamente los desarrolladores/implementadores del módulo (.cpp).

Así, por ejemplo el fichero cabecera de la biblioteca `cmath` nos dará información sobre lo que hace. Por ejemplo, si necesito el valor de la raíz cuadrada de un número, necesitaré conocer cómo se llama la función (sqrt) y el parámetro (o conjunto de parámetros que recibe), pero no me importa el algoritmo que se utiliza para calcularla. Por ejemplo desde C++11 y hasta C++23, nos encontramos con las siguientes funciones descritas en los manuales de referencia

```
/** Calcula la raiz cuadrada
* @param x número
* @return raiz cuadrada de x. Si x es negativo nos da domain error
*/
 double sqrt (double x);
 float sqrt (float x);
 long double sqrt (long double x);
 double sqrt (T x);  // additional overloads for integral types
```

El fichero .cpp asociado tendrá la implementación del algoritmo usado para su cálculo.

Como norma general, todo aquello que está en un fichero ModuloA.cpp y que no ha sido declarado en el ModuloA.h no puede ser utilizado fuera del propio fichero de implementación (.cpp). Este hecho nos guía a la hora de diseñar nuestros módulos, pues todo aquello que queramos compartir (permitimos que sea utilizado por otros módulos) lo debemos de hacer público (debe estar incluido como tal en el .h).

5.1.2. Ventajas de la compilación separada

- Los módulos se pueden reutilizar en cualquier aplicación. Nos ahorramos re-implementar. También es importante que al utilizar un módulo nos despreocupamos de los detalles de implementación. Sabemos lo que hace, pero no cómo lo hace exactamente (tampoco nos suele importar en este momento).
- Los módulos contienen funciones relacionadas desde un punto de vista lógico. Si hablamos de clases, se agrupa la clase con el conjunto de operaciones definidos para la misma.

- Nuestra aplicación puede ser desarrollada por un equipo de programadores de forma cómoda. Cada programador puede trabajar en distintos aspectos del programa, localizados en diferentes módulos, que pueden reusarse en otros programas, reduciendo el tiempo y coste del desarrollo del software.
- Tener la implementación de las funciones y clases en un fichero aparte nos permite también proteger nuestro código de cualquier tipo de manipulación y modificación por parte de terceros. De esta forma, nosotros, como empresa, podemos proporcionar una funcionalidad a un cliente y dicho cliente puede utilizar nuestro código sin tener acceso a la implementación del mismo. El cliente vería las clases y las funciones asociadas que están en el .h, lo que le permitiría utilizar el módulo correctamente, pero no sabría como están implementadas, ya que no tendrían acceso al .cpp, se le pasaría un fichero .o compilado (normalmente en forma de bibliotecas, véase la sección 6). Por tanto, mediante este esquema, podemos crear módulos que ofrezcan servicios sin que nadie pueda acceder al código y duplicar o apropiarse del activo de la empresa.
- Si modificamos un módulo, el resto de módulos que lo usen no tienen por qué ser modificados, solo recompilados.
- La compilación se puede realizar por separado. Cuando un módulo está validado y compilado no será necesario recompilarlo.

A partir de ahora, vamos a describir cómo crear un proyecto más complejo y dividirlo en múltiples ficheros con **VSCode** .

5.2. Modularización: separando el programa en múltiples ficheros

Por todo lo comentado anteriormente, separaremos el programa original `MPGeometry1` en varios ficheros, la modularización completa se detalla en los listados de la Sección 9. Se separará la implementación de las clases y funciones en cada fichero **.cpp** mientras que las declaraciones de las clases y funciones se incluirán en unos ficheros **.h** llamados ficheros de cabeceras. Para realizar este apartado crearemos un nuevo proyecto llamado **MPGeometry_MOD** donde seguiremos los siguientes pasos.

5.2.1. Crear el proyecto MPGeometry_MOD

En primer lugar, se crea el directorio `MPGeometry_MOD`. En segundo lugar, se va a crear un nuevo proyecto, por lo que a la hora de configurarlo NO se añade al espacio de trabajo actual. En la barra de menú se selecciona `File`, `Open Folder` [17]. En tercer lugar, se crea un proyecto con el nombre `MPGeometry_MOD` con apoyo del asistente `Quick Start`. Para

[17] Se puede guardar el anterior entorno de trabajo

más detalle, ver los pasos seguido en la sección 4.3. La estructura física que hemos de crear para el proyecto puede verse con detalle en la Figura 19.

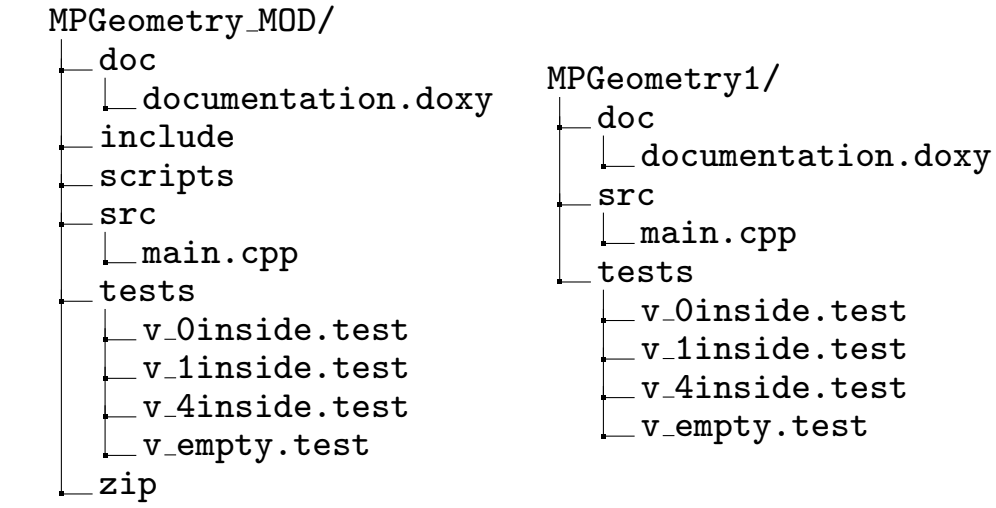

Figura 19: **a)** Estructura requerida de carpetas para la gestión de un proyecto que incluye las carpetas (**doc**, **include** (ficheros `*h`), **scripts**, **src** (ficheros `*cpp`), **tests** y **zip**). **b)** Estructura original de **MPGeometry1** ignorando las carpetas `.vscode` y `build` proveniente de la operatividad de CMake.

5.2.2. Separar declaración e implementación

El primer paso para hacer esta separación es desvincular la declaración de las clases de su implementación concreta. Para ello, dada una implementación primaria como la de la Figura 20, parte izquierda se pueden separar las implementaciones de los métodos fuera de la clase usando:

```
NombreDeLaClase :: nombreMetodo
```

como puede verse en la Figura 20.

El siguiente paso es separar aún más estos ítems: **las declaraciones** irán a un fichero con extensión **.h** y **la definición/implementación** a un fichero con extensión **.cpp**.

1. Módulo **Point2D** declarado en **Point2D.h** e implementado en **Point2D.cpp**. Contiene el código para manejar el tipo de dato **Point2D**.

2. Módulo **Rectangle** declarado en **Rectangle.h** e implementado en **Rectangle.cpp**. Contiene el código para manejar el tipo de datos **Rectangle**. Hace uso del módulo **Point2D**.

3. Módulo **main** implementado en **main.cpp**. Contiene el código que implementa el programa de cálculo de la intersección de los rectángulos y de los puntos que caen dentro de ella. Hace uso de los módulos **Point2D** y **Rectangle**.

```cpp
#include <iostream>

using namespace std;

class Helloworld {
private:
  string message;

public:
  Helloworld(){
    message="Hello world!";
  }
  void print() {
    cout << endl << message << endl;
  }
  void set(string s) {
    message = s;
  }
};


int main() {
  Helloworld hw;
  hw.set("Hola mundo!");
  hw.print();
  return 0;
}
```

```cpp
#include <iostream>
using namespace std;
class Helloworld {
private:
  string message;
public:
  Helloworld();
  void print() const;
  void set(const string &s);
};

int main() {
  Helloworld hw;
  hw.set("Hola mundo!");
  hw.print();
  return 0;
}

Helloworld::Helloworld(){
  message="Hello world!";
}
void Helloworld::print() const {
  cout << endl << message << endl;
}
void Helloworld::set(const string &s) {
  message = s;
}
```

Figura 20: Separar declaración de implementación/definición. Obsérvese a la derecha el uso del cualificador `Helloworld::` en las implementaciones de los métodos para indicar que se trata de métodos pertenecientes a esa clase, no de funciones externas.

En el fichero de cabecera (.h) irán las declaraciones de los datos y los prototipos de los métodos o funciones, esto es, todo aquello que necesita conocer cualquier módulo que haga uso de sus servicios. Pero en general, lo primero que nos vamos a encontrar en un fichero cabecera es un guardián de inclusión múltiple **(#ifndef ... #endif)**, que nos garantiza que las declaraciones dentro del fichero de cabecera **se incluirán una sola vez** en nuestro código, es decir, evitaremos las dobles inclusiones. No incluirlo es uno de los errores más comunes, como veremos posteriormente.

```
#ifndef _MODULO_H_
#define _MODULO_H_
...
#endif
```

Sustituir, `MODULO` por el identificador de la clase, o del módulo. Después de estas directivas del precompilador, aparece la definición de la clase y/o prototipos de funciones (su cabecera más punto y coma), pero **NO su implementación**. Es importante incluir en este fichero cabecera, la documentación de los distintos métodos o funciones, pues en general será consultada por los programadores que hagan uso del módulo.

Para poder compilar bien los ficheros `cpp` deberán incluirse donde sea necesario, los ficheros `.h` con la directiva:

```
#include "Point2D.h"
#include "Rectangle.h"
```

Es necesario entender el proceso de separación de *declaración* y *definición*.

```
  /**
    * @brief Queries the X coordinate
```

```
     * @return Value of X
     */
    int getX() const {
        return px;
    }
```

El modificador `const` forma parte de la declaración de un método, por lo que, los métodos **const** (de consulta), se separan como sigue:

```
/**
 * @brief Queries the X coordinate
 * @return Value of X
 */
int getX() const;
```

```
int Point2D::getX() const {
    return px;
}
```

El compilador puede distinguir entre dos métodos idénticos uno `const` y otro no *const*, por lo que si una declaración tiene const y su definición no, se produce un error.
Para facilitar la edición de ficheros, se recomienda vivamente, disponer de dos ventanas de edición en el editor, que permita mostrar en paralelo un fichero cabecera, y su homólogo fuente, como se muestra en la figura 21.

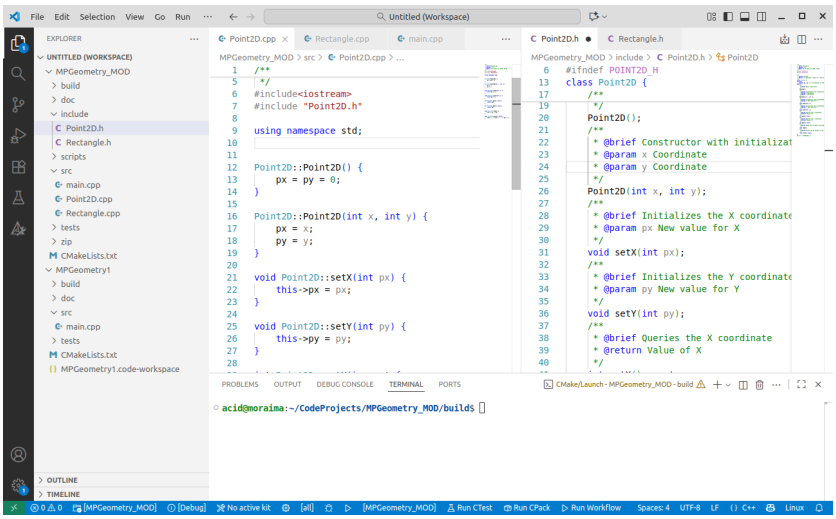

Figura 21: Comprobando correspondencia entre declaraciones y definiciones. Editando cabeceras y fuentes simultáneamente.

5.2.3. La estructura física del proyecto

Al final del proceso de edición, la estructura física del proyecto es la que se muestra en la Figura 22.

Las dos primeras subcarpetas (**.vscode** y **build**) son creadas y mantenidas automáticamente por **vscode**. El resto (**tests**, **src**, **include**) deben crearse a mano en cada proyecto con sus consabidas utilidades. Usaremos además **scripts** para almacenar scripts de gestión del mismo, y la carpeta **zip** para guardar copias de seguridad del proyecto.

Compare el contenido de los cinco ficheros, con el listado que se encuentra en el Apéndice 9, resultado de la separación por criterios funcionales y de cabeceras y fuentes.

```
./
├── .vscode
├── build
├── doc
│   └── *.doxy
├── include
│   ├── Rectangle.h
│   └── Point2D.h
├── scripts
│   └── *.sh
├── src
│   ├── main.cpp
│   ├── Point2D.cpp
│   └── Rectangle.cpp
├── tests
│   └── *.test
└── zip
    └── *.zip
```

Figura 22: Estructura básica de las carpetas de un proyecto con múltiples módulos.

5.3. Configurar CMakeLists.txt

Ya casi hemos terminado, nos queda eliminar el `main.cpp` que apareció en la raíz del proyecto y editar el fichero **CMakeLists.txt** para que **CMake** tenga en cuenta los nuevos ficheros distribuidos, a la hora de generar el binario ejecutable. El detalle de los comandos pueden encontrarse en la figura 23. En la edición es necesario incluir las modificaciones:

- Especificación de la versión de C++ 17.
- Especificación del nivel de severidad en los criterios de error del compilador Wall -Wextra -Wpedantic".
- Especificación de la lista de fuentes (con su path), que han de ser tratados en la cadena de herramientas para la obtención del ejecutable.
- Indicación del directorio donde se ubican las cabeceras. Nuevo comando `include_directories()`.

Al hacer clic en el botón **Build** (construir) en la barra de estado, **CMake** solo vuelve a compilar los **.cpp** que se han modificado, ventaja de la compilación separada, sin tener que gestionarla manualmente.

Una vez, resuelvan los errores de compilación encontrados y concluya con éxito la construcción del ejecutable, la salida en la ventana `OUTPUT` informa de los procesos seguidos por la cadena de herramientas utilizadas [18] [19]:

[18] En la GUI se denomina `Kit`.

[19] En el caso que se muestra, se usa *gnu* y *Ninja* para la construcción del ejecutable, en caso de Unix Makefile, la información es similar.

```
6  ...
7  # uso de C++ 17
8  set(CMAKE_CXX_STANDARD 17)
9
10 # uso de directivas de compilación
11 set(CMAKE_CXX_FLAGS "-Wall -Wextra -Wpedantic")
12
13 # Toma siempre la raíz como punto de referencia
14
15 # directorio donde buscar los *h
16 include_directories(include)
17
18 # Objetivo MPGeometry_MOD con los tres fuentes
19 add_executable(MPGeometry_MOD src/Point2D.cpp src/Rectangle.cpp src/main.cpp)
```

Figura 23: Extracto de `CMakeLists.txt` con la corrección y nuevas especificaciones.

```
[main] Building folder: /home/acid/CodeProjects/MPGeometry_MOD/build all
[build] Starting build
[proc] Executing command: /usr/bin/cmake --build /home/acid/CodeProjects/MPGeometr
[build] [ 25%] Building CXX object CMakeFiles/MPGeometry_MOD.dir/src/main.cpp.o
[build] [ 50%] Building CXX object CMakeFiles/MPGeometry_MOD.dir/src/Point2D.cpp.o
[build] [ 75%] Building CXX object CMakeFiles/MPGeometry_MOD.dir/src/Rectangle.cpp
[build] [100%] Linking CXX executable MPGeometry_MOD
[build] [100%] Built target MPGeometry_MOD
[driver] Build completed: 00:00:00.286
[build] Build finished with exit code 0
```

En la salida podemos observar que se compilan uno a uno cada fuente por separado (con la obtención de los artefactos binarios (*.o)), luego se enlaza en el ejecutable (target) **MPGeometry_MOD** ubicado en `build`. Para proceder a su ejecución podemos elegir los mismos procedimientos que en versiones anteriores. Desde el terminal usando un fichero como entrada como `v_0inside.test`.

```
~/CodeProjects/MPGeometry_MOD/build$ ./MPGeometry_MOD < ../tests/v_0inside.test
```

Otra alternativa es mediante la selección de algún `launch` (inicio), que requiere de datos de entrada desde el teclado. Se sugiere explorar la vista activa seleccionado `CMake` en el Panel de Actividades) donde se ofrecen nuevas formas de lanzamiento o inicio.

6. Bibliotecas

Mediante **CMake** un proyecto puede definir más de un ejecutable, basta con incluir más de un objetivo mediante el comando `add_executable`. Por ejemplo, en un proyecto podríamos tener `main1.cpp` y `main2.cpp` y generar dos ejecutables. Si cada main utiliza las clases `Point2D` y `Rectangle`, CMake genera reglas que hacen que los módulos de las clases se compilen dos veces (una para cada ejecutable). Para optimizar la compilación y facilitar la reutilización, vamos a crear una biblioteca (que contendrá ambas clases). Esta biblioteca podrá ser enlazada posteriormente con cualquier función main que la necesite.

Para este proceso vamos a crear un nuevo proyecto con ayuda del Quick Start, por lo que comenzamos con un WorkSpace (espacio de tra-

```
cmake_minimum_required(VERSION 3.10.0)
project(MPGeometry_LIB VERSION 0.1.0 LANGUAGES C CXX)

# uso de C++ 17
set(CMAKE_CXX_STANDARD 17)

# uso de directivas de compilación ampliadas
set(CMAKE_CXX_FLAGS "-Wall -Wextra -Wpedantic")

set(SOURCE_DIR src)
set(INCLUDE_DIRECTORIES include)
include_directories(${INCLUDE_DIRECTORIES})

add_library(formas ${SOURCE_DIR}/Point2D.cpp ${SOURCE_DIR}/Rectangle.cpp )
```

Figura 24: Detalle de `CMakeLists.txt` con los comandos para la creación de una biblioteca y el ajuste de las especificaciones del proyecto.

bajo) vacío [20].

En lugar de crear un nuevo directorio y partir de cero, vamos a copiar el directorio `MPGeometry_MOD` en un nuevo `MPGeometry_LIB` y a reutilizar los ficheros cabeceras y fuentes de las clases `Point2D` y `Rectangle`. Eliminamos el fichero `CMakeLists.txt` y el fuente `main.cpp`. La estructura del directorio es idéntica a la Figura 22 con un fuente menos.

Ahora sí, creamos el proyecto como en la sección 4.3 con una diferencia, el objetivo ahora es crear una biblioteca (library). Procedemos como sigue: en la barra de menú, se selecciona `View`, `Command Palette`, `Quick Start` y seleccionamos las opciones que se detallan a continuación: (1) el nombre del proyecto es `MPGeometry_LIB`, (2) se trata de código `C/C++` y (3) seleccionamos `Library create a library`.

Al igual que ocurriera antes, las **CMake Tools** han generado varios ficheros y carpetas, entre otros un `CMakeLists.txt` nuevo. Procedemos a editarlo para personalizar nuestro proyecto con el siguiente contenido:

Podemos identificar el comando `add_library()` donde se especifica el nombre de la biblioteca, se ha elegido `formas` y se construye a partir de los fuentes **Point2D.cpp** y **Rectangle.cpp** una vez especificada la carpeta donde se encuentran las cabeceras con el comando `include_directories()`, como hicieramos en proyectos anteriores. La novedad es que hemos usado variables como `SOURCE_DIR` e `INCLUDE_DIRECTORIES` que se asignan con el comando `set()` y cuyo valor se usa mediante `${nombreVariable}`. En la ventana OUTPUT pueden verse los pasos seguidos durante la construcción de la biblioteca estática.

```
[main] Building folder: /home/acid/CodeProjects/MPGeometry_LIB/build all
[build] Starting build
[proc] Executing command: /usr/bin/cmake --build
                                /home/acid/CodeProjects/MPGeometry_LIB/build
                                    --config Debug --target all -j 20 --
[build] [ 33%] Building CXX object CMakeFiles/formas.dir/src/Point2D.cpp.o
[build] [ 66%] Building CXX object CMakeFiles/formas.dir/src/Rectangle.cpp.o
[build] [100%] Linking CXX static library libformas.a
[build] [100%] Built target formas
[driver] Build completed: 00:00:00.277
[build] Build finished with exit code 0
```

[20] De no hacerse así, se entiende que se trata de un proyecto más complejo y comparte un `CMakeLists.txt` anterior.

```
./
├── doc
│   └── *.doxy
├── include
│   ├── Rectangle.h
│   └── Point2D.h
├── lib
│   └── libformas.a
├── scripts
│   └── *.sh
├── src
│   └── main.cpp
├── tests
│   └── *.test
└── zip
    └── *.zip
```

Figura 25: Estructura física del proyecto de aplicación. Observe la carpeta **lib** que contiene una copia de la biblioteca creada en el anterior proyecto, **MPGeometry_LIB**, y en **include** una copia de las cabeceras públicas de la biblioteca.

Con ello ya disponemos de una colección de archivos de código objeto precompilados bajo el nombre `libformas.a`, disponible en el directorio `buid`.

6.1. Usar la biblioteca en un proyecto

Siguiendo el ejemplo de la intersección de rectángulos, ahora hemos de poder integrar la biblioteca con un programa principal en un ejecutable final mediante el proceso de enlazado. El nuevo proyecto llamado `MPGeometry_APP` debe constar de constar sólo del fichero fuente **main.cpp**, el resto desaparecen del proyecto. Existen diferentes formas de utilizar una biblioteca, la que vamos a mostrar es sin vinculación entre el proyecto de la aplicación y el proyecto que genera la biblioteca [21]. La estructura física del proyecto `MPGeometry_APP` se detalla en la Figura 25.

Para poder usar una biblioteca propia o externa es necesario indicar a **CMake** en primer lugar, dónde se encuentra la biblioteca con el comando `link_directories()` y en segundo lugar, qué objetivo tiene que enlazar con la biblioteca mediante el comando `target_link_libraries()`. Como conclusión, mostramos el contenido completo del fichero de configuración para CMake.

[21] Lo que implica que si se hace una modificación en los fuentes de la biblioteca, la aplicación no se da por enterada, pues se linka con el fichero (.a) original.

```
cmake_minimum_required(VERSION 3.10.0)
project(MPGeometry_APP VERSION 0.1.0 LANGUAGES C CXX)

# uso de C++ 17
set(CMAKE_CXX_STANDARD 17)

# uso de directivas de compilación
# -Wall aviso de warnings,
# -Wextra aviso de errores más sutiles
# -Wpedantic uso de ansi C++
#set(CMAKE_CXX_FLAGS "-Wall -Wextra -Wpedantic")

set(LIBRARY_DIR lib) # ruta donde se encuentra el archivo .a
set(SOURCE_DIR src)
set(INCLUDE_DIRECTORIES include)

include_directories(${INCLUDE_DIRECTORIES})
add_executable(MPGeometry_APP  ${SOURCE_DIR}/main.cpp)

# Indica dónde buscar los archivos de la biblioteca (.a)
link_directories(${LIBRARY_DIR})

# Enlaza la biblioteca por su nombre
# Si el archivo se llama 'libformas.a', el nombre es 'formas'
target_link_libraries(MPGeometry_APP PUBLIC formas)

#o bien de forma alternativa con el path completo
#target_link_libraries(MPGeometry_APP  PUBLIC "${CMAKE_SOURCE_DIR}/lib/libformas.a")
```

Figura 26: Edición parcial de `CMakeLists.txt` con los comandos necesarios para la explotación de la biblioteca.

7. Personalización de VSCode

7.1. Los LLMs, uso y abuso

Que los LLM, Large Language Models [22], son herramientas muy potentes en una programación básica o intermedia como la que se va a desarrollar en estas prácticas [23], es un hecho indudable, y es todo *un reto saber cómo su uso puede* **lesionar** *o* **beneficiar** *el aprendizaje de la programación para el estudiante*.

Algunas de las ayudas que puede encontrar son:

1. Autocompletado de código, además de sugerencias contextuales mientras se escribe.
2. Generación de funciones, clases y fragmentos de código a partir de comentarios o descripciones.
3. Explicación personalizada de conceptos.
4. Explicaciones de código y comentarios: resúmenes, paso a paso y anotaciones.
5. Explicación/justificación de errores.

Se recomienda acudir preferentemente al profesorado para la aclaración y/o explicación de dudas acerca de las prácticas pues, las pistas y recomendaciones supondrán un **aprendizaje significativo** frente a la información dada por un LLM.

[22] chatGPT, Gemini, Codex el modelo LLM que alimenta a Copilot, etc.

[23] Tiene unas tasa de corrección de alrededor del 70 %-80 % en problemas introductorios ref https://arxiv.org/pdf/2304.02491.

En caso de consultarse un LLM, las ayudas tipo (1.) o (2.) no son aceptables pues, provocan una dependencia excesiva de los LLMs con la elaboración directa de soluciones y **no implican aprendizaje significativo** alguno. Las ayudas aceptables son (3.), (4.) y (5.). Para beneficiarse de explicaciones, guías y soporte sin caer en la generación de código automático, hay que tomar precauciones. En un prompt es necesario indicar que, no se desea código, sino solo explicaciones.

Se debe evitar que el agente inserte/edite código en los archivos mientras está programando [24]. La extensión `GitHub Copilot` (pair AI) **no debe estar activa**.

7.2. Unos scripts

VSCode puede ejecutar scripts del sistema operativo (Linux) para realizar tareas externas a la propia GUI, normalmente para operaciones de mantenimiento del proyecto o para llamar a programas externos. En esta asignatura se propone el uso de los siguientes scripts, los cuales se encuentran disponibles en la carpeta *CodeProjects* y que es aconsejable tenerlos reunidos en una misma carpeta, por ejemplo **scripts**. Son los que aparecen a continuación (los cuales pueden ser modificados o ampliados a criterio del alumno).

runUpdate.sh Configura parámetros básicos y las carpetas del proyecto. Esta script hay que usarla siempre la primera vez que se abre el proyecto.

runDocumentation.sh Según el fichero **.doxy** que se haya guardado en la carpeta **doc**/ del proyecto, genera la documentación de Doxygen de forma automática y en formato **html** y **latex**. Si la carpeta no existe, se crea automáticamente.

runZipProject.sh Genera un zip comprimido de las carpetas más importantes del proyecto para llevarlo a otro ordenador o para entregar las prácticas en Prado. Coloca el fichero zip en la carpeta **zip**/ y, si no existe, la crea.

runTest.sh Realiza un conjunto de tests de integridad mínimos con los que comprobar la ejecución correcta con un conjunto de datos suministrados. La salida sale por pantalla.

Para ejecutarlas desde **VSCode** , basta hacer click con el botón derecho del ratón en la que queramos ejecutar y seleccionar “Run”.

[24] De esta forma se evita el modo de pastoreo. Si el agente es el pastor, adivina quién es la oveja...

8. El programa original

```
#include <iostream>
#include <cmath>
using namespace std;
class Point2D {
private:
    double px, py;
public:
    Point2D() {
        px = py = 0;
    }
    Point2D(int x, int y) {
        px = x;
        py = y;
    }
    void setX(int px) {
        this->px = px;
    }
    void setY(int py) {
        this->py = py;
    }
    int getX() const {
        return px;
    }
    int getY() const {
        return py;
    }
    void read() {
        cin >> px >> py;
    }
    void print() const {
        cout << "("<<px << "," << py<<")";
    }
};

class Rectangle {
private:
    Point2D topleft, bottomright; ///< Points that define the rectangle
public:
    Rectangle() { }
    Rectangle(int x, int y, int w, int h) {
        topleft.setX(x);
        topleft.setY(y);
        bottomright.setX(x+w-1);
        bottomright.setY(y+h-1);
        // setGeometry(x,y,w,h);
    }
    void setGeometry(int x, int y, int w, int h) {
        topleft.setX(x);
        topleft.setY(y);
        bottomright.setX(x+w);
        bottomright.setY(y-h);
    }
    void setGeometry(const Point2D &tl, const Point2D &br) {
        topleft = tl;
        bottomright = br;
    }
    Point2D getTopLeft() const {
        return topleft;
    }
    Point2D getBottomRight() const {
        return bottomright;
    }
    bool isEmpty() const {
        return topleft.getX()>bottomright.getX() || topleft.getY() < bottomright.getY();
    }
    void read() {
        topleft.read();
        bottomright.read();
    }
    void print() const {
        cout << "[";
        topleft.print();
        cout << "␣-␣";
        bottomright.print();
        cout << "]␣";
    }
    friend Rectangle doOverlap(const Rectangle & r1, const Rectangle &r2);

};


Rectangle doOverlap(const Rectangle & r1, const Rectangle &r2) {
    Rectangle result;
    Point2D rTL, rBR;
    rTL.setX(max(r1.topleft.getX(),r2.topleft.getX()));
    rTL.setY(min(r1.topleft.getY(),r2.topleft.getY()));
    rBR.setX(min(r1.bottomright.getX(),r2.bottomright.getX()));
    rBR.setY(max(r1.bottomright.getY(),r2.bottomright.getY()));
    result.setGeometry(rTL,rBR);
    return result; // Read more
}

bool isInside(const Point2D &p, const Rectangle &r) {
    return r.getTopLeft().getX() <= p.getX() && p.getX()<=r.getBottomRight().getX() &&
           r.getTopLeft().getY() >= p.getY() && p.getY()>=r.getBottomRight().getY();
}
```

```
int main() {
    Rectangle A, B, Intersection;
    Point2D p;
    int count;

    A.setGeometry(2,5,3,3);
    cout << "First rectangle is ";
    A.print();
    cout << endl << "Type second rectangle: ";
    B.read();
    cout << endl << "Calculating intersection of: ";
    A.print();
    cout << " and ";
    B.print();
    cout << endl;
    Intersection = doOverlap(A,B);
    if (Intersection.isEmpty()) {
        cerr << "Empty intersection"<<endl;
    } else {
        cout << "The intersection is: ";
        Intersection.print();
        count = 0;
        cout << endl << "Reading points...";
        p.read();
        while (p.getX()>=0 && p.getY()>=0) {
            if (isInside(p,Intersection)) {
                p.print();
                count ++;
            }
            p.read();
        }
        if (count > 0)
            cout << " fall within the intersection ("<< count<<" total)" << endl;
        else
            cout << " None of them falls within the intersection "<<endl;
    }
    return 0;
}
```

8.1. Fichero de validación `v_0inside.test`

El cuadro siguiente muestra los datos de un fichero de validación hasta el valor “-1 0” que concluye los datos de entrada. El resto del fichero se proporciona para automatizar los procesos de validación tal y como se indica en el script **doTests.sh** (Sección 7).

```
4 3 8 0
0 0
1 1
-1 0

%%%CALL < tests/v_0inside.test
%%%OUTPUT
First rectangle is [(2,5) - (5,2)]
Type second rectangle:
Calculating intersection of: [(2,5) - (5,2)]  and [(4,3) - (8,0)]
The intersection is: [(4,3) - (5,2)]
Reading points... None of them falls within the intersection
```

9. Apendice 1. Modularización

9.1. Point2D.h

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D {
private:
    double px, py;
public:
    Point2D();
    Point2D(int x, int y);
    void setX(int px);
    void setY(int py);
    int getX() const;
    int getY() const;
    void read();
    void print() const;
};
#endif
```

9.2. Rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

#include "Point2D.h"

class Rectangle {
private:
    Point2D topleft, bottomright;
public:
    Rectangle();
    Rectangle(int x, int y, int w, int h);
    void setGeometry(int x, int y, int w, int h);
    void setGeometry(const Point2D &tl, const Point2D &br);
    Point2D getTopLeft() const;
    Point2D getBottomRight() const;
    bool isEmpty() const;
    void read();
    void print() const;
    friend Rectangle doOverlap(const Rectangle & r1, const Rectangle &r2);
};
bool isInside(const Point2D &p, const Rectangle &r);
#endif
```

9.3. Point2D.cpp

```
#include<iostream>
#include "Point2D.h"

using namespace std;

Point2D::Point2D() {
    px = py = 0;
}
Point2D::Point2D(int x, int y) {
    px = x;
    py = y;
}
void Point2D::setX(int px) {
    this->px = px;
}
void Point2D::setY(int py) {
    this->py = py;
}
int Point2D::getX() const {
    return px;
}
int Point2D::getY() const {
    return py;
}
void Point2D::read() {
    cin >> px >> py;
}
void Point2D::print() const {
    cout << "("<<px << "," << py<<")";
}
```

9.4. Rectangle.cpp

```
#include<iostream>
#include<cmath>
#include "Point2D.h"
#include "Rectangle.h"
using namespace std;

Rectangle::Rectangle() { }

Rectangle::Rectangle(int x, int y, int w, int h) {
    topleft.setX(x);
    topleft.setY(y);
    bottomright.setX(x+w-1);
    bottomright.setY(y+h-1);
    // setGeometry(x,y,w,h);
}

void Rectangle::setGeometry(int x, int y, int w, int h) {
    topleft.setX(x);
    topleft.setY(y);
    bottomright.setX(x+w);
    bottomright.setY(y-h);
}

void Rectangle::setGeometry(const Point2D &tl, const Point2D &br) {
    topleft = tl;
    bottomright = br;
}

Point2D Rectangle::getTopLeft() const {
    return topleft;
}

Point2D Rectangle::getBottomRight() const {
    return bottomright;
}

bool Rectangle::isEmpty() const {
    return topleft.getX()>bottomright.getX() || topleft.getY() < bottomright.getY();
}

void Rectangle::read() {
    topleft.read();
    bottomright.read();
}

void Rectangle::print() const {
    cout << "[";
    topleft.print();
    cout << "␣-␣";
    bottomright.print();
    cout << "]␣";
}

Rectangle doOverlap(const Rectangle & r1, const Rectangle &r2) {
    Rectangle result;
    Point2D rTL, rBR;
    rTL.setX(max(r1.topleft.getX(),r2.topleft.getX()));
    rTL.setY(min(r1.topleft.getY(),r2.topleft.getY()));
    rBR.setX(min(r1.bottomright.getX(),r2.bottomright.getX()));
    rBR.setY(max(r1.bottomright.getY(),r2.bottomright.getY()));
    result.setGeometry(rTL,rBR);
    return result; // Read more
}

bool isInside(const Point2D &p, const Rectangle &r) {
    return r.getTopLeft().getX() <= p.getX() && p.getX()<=r.getBottomRight().getX() &&
           r.getTopLeft().getY() >= p.getY() && p.getY()>=r.getBottomRight().getY();
}
```

9.5. main.cpp

```
#include <iostream>
#include <cmath>
#include "Point2D.h"
#include "Rectangle.h"

using namespace std;

int main() {
    Rectangle A, B, Intersection;
    Point2D p;
    int count;

    A.setGeometry(2,5,3,3);
    cout << "First rectangle is ";
    A.print();
    cout << endl << "Type second rectangle: ";
    B.read();
    cout << endl << "Calculating intersection of: ";
    A.print();
    cout << " and ";
    B.print();
    cout << endl;
    Intersection = doOverlap(A,B);
    if (Intersection.isEmpty()) {
        cerr << "Empty intersection"<<endl;
    } else {
        cout << "The intersection is: ";
        Intersection.print();
        count = 0;
        cout << endl << "Reading points...";
        p.read();
        while (p.getX()>=0 && p.getY()>=0) {
            if (isInside(p,Intersection)) {
                p.print();
                count ++;
            }
            p.read();
        }
        if (count > 0)
            cout << " fall within the intersection ("<< count<<" total)" << endl;
        else
            cout << " None of them falls within the intersection "<<endl;
    }

    return 0;
}
```

10. Apéndice 2: Errores típicos al modularizar un programa

Muchas veces al modularizar un programa suelen aparecer múltiples errores de compilación, que en su mayor parte son debidos a una mala definición del contenido de los ficheros cabecera (.h). ¿Por qué? Pues es simple, el resto de los módulos hacen uso de las funcionalidades de un módulo A realizando únicamente `#include "A.h"` y es el compilador/enlazador (lo veremos después) el que se encarga de hacer que todo sea correcto, por tanto un error en el diseño del fichero `A.h` se puede propagar hacia otros módulos, aunque en algunas situaciones parezca que todo es correcto.

Pasamos a ver una lista de errores típicos que se pueden (suelen) cometer

#1 Todo fichero .cpp asociado a un .h debe comenzar con la directiva `#include "xxxx.h"`

```
(A.h)
  .....
  class A{
   public:
     ...
     f1();
 };
=========
(util.h)
 const double PI = 3.14;
 double sqrtXPI( );
```

```
(A.cpp)
void A::f1(){ ... }
// Error no reconoce A
======
(util.cpp)
funcionX( ) { ...}
//OK, compila
funcionXPI() {
   return 2*PI;
    // Error No reconoce PI
}
```

#2 No utilizar el guardián de inclusión múltiple: Cuando el preprocesador encuentra un #include, sustituye la línea con el contenido del fichero. Esto puede generar un problema en el caso en el que el fichero se haya incluido múltiples veces durante el proceso de compilación.

```
(A.h)
  # .....
  class A{
   public:
    ...
 };
=========
(B.h)
 #include "A.h"
   class B {
    private:
      A x;
    ..
};
```

```
(main.cpp)
 #include "B.h"
 #include "A.h"
 int x;
 ======
(lo que se obtiene al sustituir en main.cpp)
     class A {  };
     class B {  };
     class A {  };  // Redefinicion de A
       public:
         ...
     } int x;
```

Para evitarlo se utiliza una etiqueta guardián en cada fichero

```
(AAA.h)
#ifndef AAA_H
#define AAA_H

\\el contenido del fichero

#endif
```

Así, cuando se incluye por primera vez el fichero se define la etiqueta `AAA_H`, y el resto de veces al encontrarla definida no se vuelve a incluir. Como norma, la etiqueta coincide con el nombre del fichero, pues se asume que es indicativo de la funcionalidad del mismo, toda en mayúsculas y terminada con el sufijo `_H`

#3 Olvidar el punto y coma al definir la clase, como nuestra el ejemplo. En este caso, el error nos aparece cuando tratamos de compilar el fichero main.cpp, aunque dicho fichero sea correcto. El problema se ve claramente cuando sustituimos el include de `A.h` por su contenido.

```
(A.h)
# .....
class A{
  public:
    ...
}
```

```
=========
 (main.cpp)
 #include "A.h"
 int x;
 =======
(lo que se obtiene al sutituir en main.cpp)
     class A{
       public:
         ...
     } int x;
```

#4 Un programa hace uso de una funcionalidad no especificada en el fichero cabecera, .h, como se nos muestra en el ejemplo con la función `funcionY()` definida sólo en el ámbito del fichero A.cpp

```
(A.h)
class A {
  public:
   bool m1();
  private:
   int  m2();
  };
void funcionX();
=======
(A.cpp)
#include "A.h"
void funcionY() { // OK
   ...
   funcionX(); //OK
   // por incluir A.h
   ...
};
void funcionX()
   { ... }; //OK
...
```

```
=========
 (main.cpp)
 #include "A.h"
 int main(){
 A x;
  x.m1(); // correcto
  x.m2(); // incorrecto, es privada
  funcionX(); // correcto
  funcionY(); // incorrecto, no definida
              // en A.h
}
```

#5 Implementar los métodos de una clase sin utilizar el operador de ámbito

```
(A.h)
class A {
  public:
     A();
     void f0();
     int f1();
     bool f4();
     private:
     bool m2();
  };
// Funciones aux
void f2();
bool f3()
```

```
========
// Fichero de implementacion
// de la clase A
(A.cpp)
#include "A.h"
  A::A(){  ... }; // OK
  void A::f0() { ...} // OK
  int A::f1() { ... } // OK
  bool A::m2() {....} //OK
  void f2(){ ... } // OK
  bool f3(){ ... } // OK
  bool f4() {....} // Incorrecto
  // Hay que especificar ambito
  // bool A::f4() { ....}
     ...
};
funcionX() { ... };
```

#6 Es conveniente no utilizar un `using namespace` en los ficheros .h

11. Videotutoriales

1. Instalación y preparación del entorno de trabajo (Abrir →) en `Tutorial for Beginners (For Absolute Beginners)` primeros 8mn. La segunda mitad del vídeo se dedica al depurador, materia de otro guion.

2. Tutorial de VSCode instalación, extensiones y configuración de fondos, Themes(Abrir →)