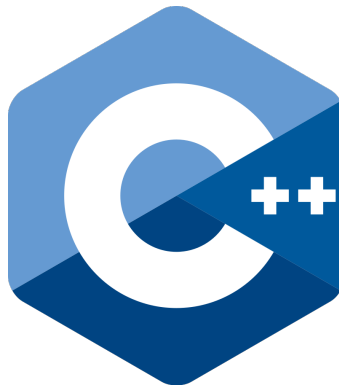




# Metodología de la Programación

DGIM

Curso 2021/2022



## Guion de prácticas

*Entorno de desarrollo de software con NetBeans*

*Febrero de 2021*



# Índice

<b>1. Introducción</b>	<b>6</b>
<b>2. Empezando</b>	<b>6</b>
<b>3. Un primer proyecto</b>	<b>8</b>
3.1. Creación del proyecto NetBeans	8
3.2. Creación de un fichero fuente	9
3.3. Generando y ejecutando el binario	9
<b>4. Un proyecto de compilación separada</b>	<b>11</b>
4.1. El programa en un único fichero	11
4.2. Ejecutando el programa	11
4.3. Utilizando ficheros de entrada	11
4.4. Compilación separada	13
4.4.1. Ocultamiento de Información	14
4.4.2. Ventajas de la compilación separada	14
4.5. Modularización: separando el programa en múltiples ficheros	15
4.5.1. Separar declaración e implementación	15
4.6. Crear la estructura del proyecto en la vista física	17
4.7. Configurar la vista lógica del proyecto	18
4.8. Configurar los parámetros del proyecto	18
<b>5. Bibliotecas</b>	<b>20</b>
5.1. Crear la estructura de ficheros	20
5.2. Usar la nueva biblioteca en el proyecto	21
<b>6. Depurador</b>	<b>24</b>
6.1. Conceptos básicos	24
6.2. Ejecución de un programa paso a paso	24
6.3. Inspección y modificación de datos	25
6.4. Punto de ruptura condicional	26
6.5. Inspección de la pila	26
6.6. Reparación del código	27
<b>7. Otras características de NetBeans</b>	<b>27</b>
<b>8. Personalización de NetBeans</b>	<b>30</b>
<b>9. El programa original</b>	<b>31</b>
9.1. Fichero de validación <code>v_0inside.test</code>	32
<b>10. Apendice 2. Modularización</b>	<b>33</b>
10.1. <code>Point2D.h</code>	33
10.2. <code>Rectangle.h</code>	33
10.3. <code>Point2D.cpp</code>	33
10.4. <code>Rectangle.cpp</code>	34
10.5. <code>main.cpp</code>	35



<b>11. Apéndice 2: Errores típicos al modularizar un programa</b>	<b>35</b>
---	-----------

<b>12. Videotutoriales</b>	<b>37</b>
----------------------------	-----------





## 1. Introducción

Esta sesión de prácticas está dedicada a aprender a utilizar el entorno de desarrollo de software multi-language **NetBeans** como alternativa a la gestión de proyectos desde la línea de comandos. NetBeans es un entorno de desarrollo integrado libre y multiplataforma, creado principalmente para el lenguaje de programación Java, pero que ofrece soporte para otros muchos lenguajes de programación. Existe además un número importante de módulos para extenderlo. NetBeans es un producto libre y gratuito sin restricciones de uso. En este guión se explicará cómo utilizarlo en un **Linux** que ya tenga instalado **g++** y **make**. Si aún no lo tiene instalado, puede ver la sección 4 de la guía de configuración con VirtualBox y GNU/Linux proporcionado en **prado**.

## 2. Empezando

Es necesario instalar NetBeans con JDK <sup>1</sup> y el plugin para **C++** <sup>2</sup> instalados en ese orden. También se puede instalar el plugin de **C++** desde el menú 'Tools-Plugins' del menu de NetBeans.

Una vez instalado se podrá ejecutar el programa 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 (para la versión 8.2 disponible en las aulas de prácticas)

```
/usr/local/netbeans-8.2/bin/netbeans
```

o simplemente (para la versión 12.2 si la habéis instalado recientemente):

```
netbeans
```

En la Figura 1 se puede ver la distribución de las áreas de trabajo en NetBeans:

- La parte superior contiene un menú de opciones típico de un entorno de desarrollo de software del que se irá detallando lo más importante a lo largo de este guión. En cualquier caso, para más información se puede consultar la guía oficial de NetBeans<sup>3</sup>.
- El cuadrante superior izquierdo muestra el navegador de proyectos.
- El cuadrante superior derecho muestra las pestañas para editar ficheros fuente.
- El cuadrante inferior izquierdo, que muestra el contexto del código (funciones, pila, variables locales) durante las sesiones de depuración.

---

<sup>1</sup>([Abrir →](#))

<sup>2</sup>([Abrir →](#))

<sup>3</sup>([Abrir →](#))

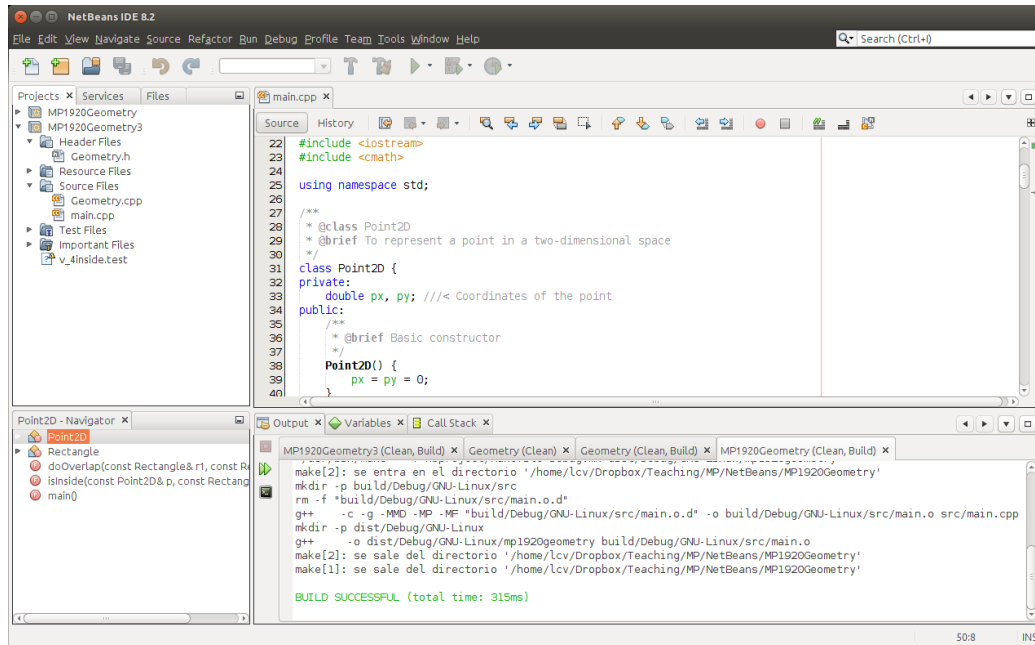


Figura 1: Entorno de trabajo en NetBeans con C++

- El cuadrante inferior derecho que muestra múltiples pestañas asociadas con la ejecución del proyecto:
  - Output. Muestra las salidas estándar y de error y recoge la entrada estándar.
  - Terminal. Línea de comandos en la carpeta principal del proyecto.
  - Variables. Inspección de variables durante la depuración.
  - Call Stack. Estado de la pila de llamadas (depuración).
  - Breakpoints. Lista de puntos de ruptura activos (depuración).

## 3. Un primer proyecto

### 3.1. Creación del proyecto NetBeans

En el navegador de proyecto (cuadrante superior izquierda) con la pestaña “Projects” activa, pulsar con el botón derecho y seleccionar “New Project” (alternativamente “File” - “New Project” o pulsar el icono de la carpeta con un +). Seleccionar el tipo de aplicación que se quiere construir (Figura 3.a), entre los múltiples lenguajes soportados por NetBeans, en este caso “C/C++ Application”.

Si no apareciese el lenguaje C++, habrá que instalarlo desde el menú superior **Tools** -> **Plugins**. En la pestaña **Settings** tendremos que asegurarnos de que estén señaladas todas las opciones (ver Figura 2) y a continuación ir a la pestaña **Available Plugins**, darle al botón **Check for Newest** y seleccionar C++ en la lista. Seguir el proceso de instalación y reiniciar NetBeans.

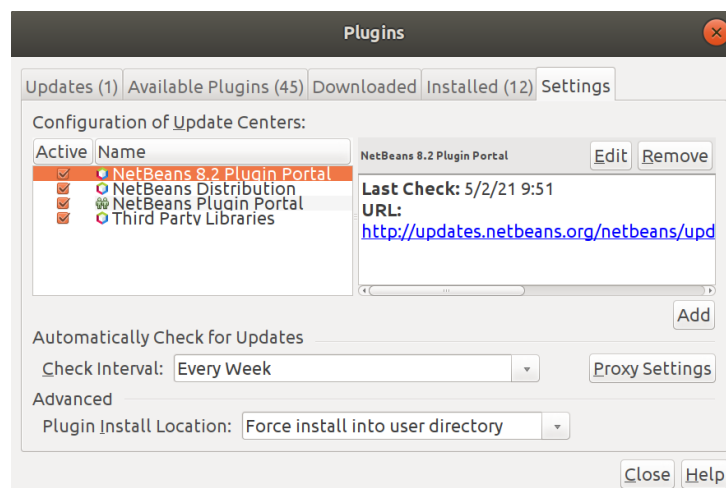


Figura 2: Instalando plugin para el lenguaje C++

Seleccionar el nombre del proyecto (“NuevoProyecto”) y la ubicación que se le quiere dar en disco “Project location / Browse” (Figura 3.b).

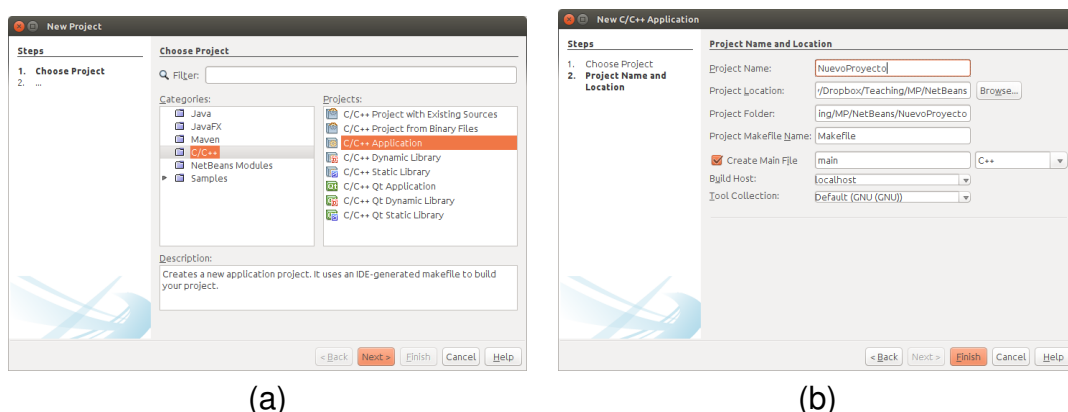


Figura 3: Creando un proyecto nuevo



En el navegador de proyectos (cuadrante superior izquierdo) aparecerán los árboles lógicos y físicos del proyecto recién creado (Figura 4).

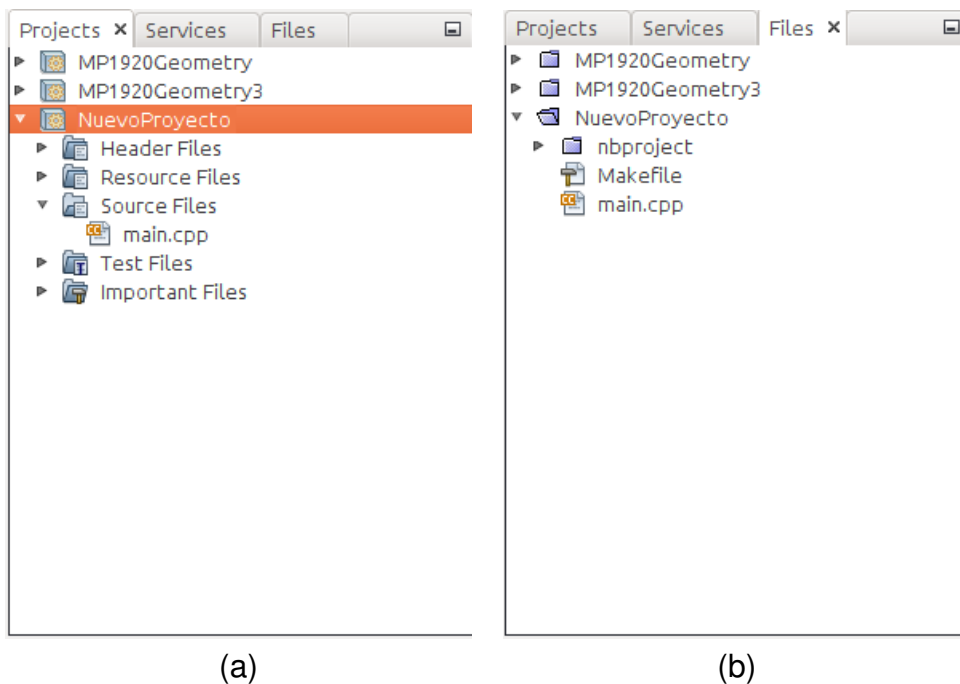


Figura 4: Árbol lógico (izquierda: pestaña “Projects”) y físico (derecha: pestaña “Files”) de un proyecto nuevo

### 3.2. Creación de un fichero fuente

Vamos a crear el primer fichero fuente del proyecto. Para ello, en el recién creado de forma automática (ver tick señalado en la Figura 3.b) fichero **main.cpp** se introduce un primer programa en C++, como el conocido “Hola Mundo”<sup>4</sup> (Figura 1).

Si no estuviese creado, se podría ir al menú “File”-“New File” y se selecciona “C++ Source File” (Figura 5).

### 3.3. Generando y ejecutando el binario

Desde NetBeans se puede compilar el proyecto y ejecutarlo en un mismo paso:

#### Run - Clean and Build Project

Cuya salida aparece en el cuadrante inferior derecho, pestaña **Output** (Figura 5).

El binario recién generado se encuentra en la carpeta **dist** tal y como muestra el cuadrante superior derecho de la Figura 5.

La ejecución del binario que se acaba de generar se ordena como

#### Run - Run Project

Y se puede seguir su ejecución, desde la pestaña **Output** (Figura 6) como si fuese una consola de órdenes del sistema operativo.

<sup>4</sup>El programa “Hola mundo” en más de 200 lenguajes de programación ([Abrir →](#))

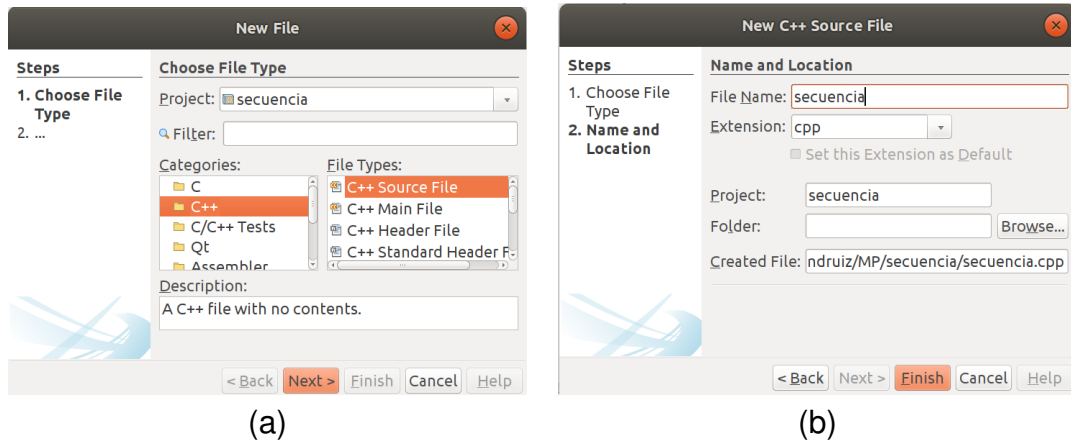


Figura 5: Creación de un nuevo fichero fuente en el proyecto

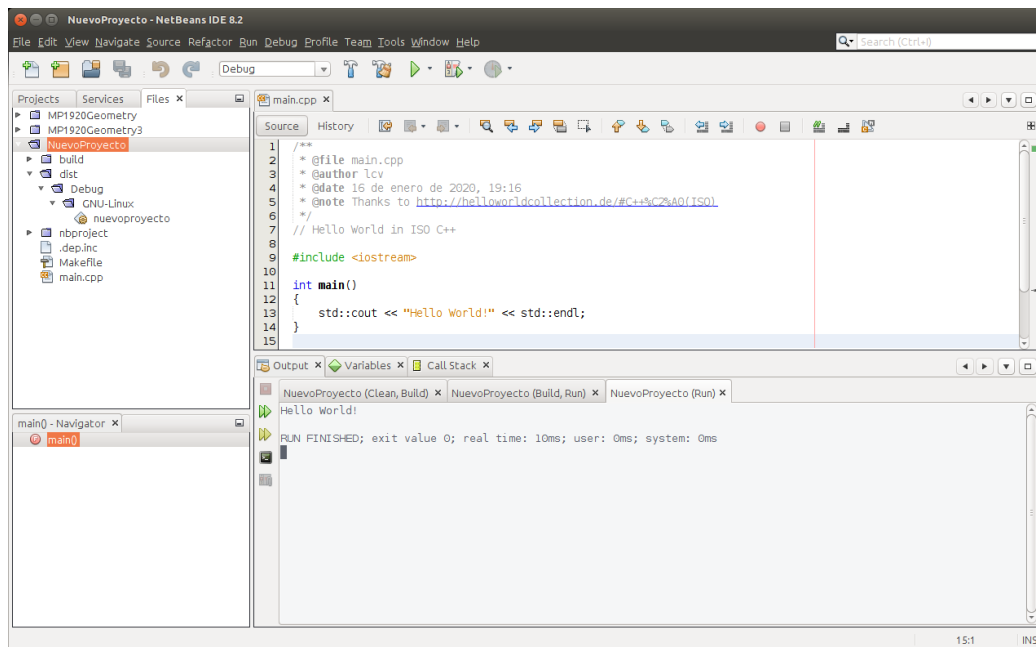


Figura 6: Ejecutando el binario

## 4. Un proyecto de compilación separada

El ejemplo desarrollado en esta sección se encuentra comentado en dos videotutoriales, la versión básica ([Abrir →](#)) y avanzada ([Abrir →](#))

### 4.1. El programa en un único fichero

El objetivo del programa es calcular la intersección de dos rectángulos, leer una serie de puntos y calcular cuántos de esos puntos caen dentro de la intersección. Para ello se utilizan las clases **Point2D** y **Rectangle** que aparecen en el código escrito en un único fichero mostrado en la sección 9. Crear un nuevo proyecto llamado **MPGeometry** y copiar este código en el fichero **main.cpp**. Desde la vista de ficheros, crear una carpeta llamada **src** y mover el fichero dentro de esta carpeta.

Desde la vista de ficheros, crear una carpeta dando click con el botón derecho del ratón al proyecto **New → Folder...** que llamaremos **src** (ver Figura 7) y moveremos el fichero **main.cpp** dentro de esta carpeta.

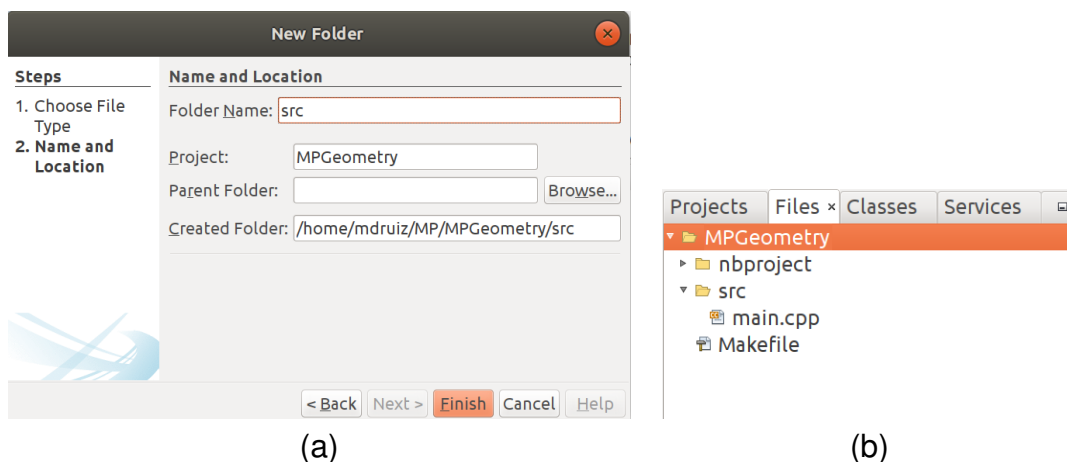


Figura 7: Creación de una carpeta “src” para el proyecto **MPGeometry**

### 4.2. Ejecutando el programa

Ejecutar el programa como se ha visto hasta ahora y seguir la secuencia que se muestra en la Figura 8.a) introduciendo los mismos datos. Otra forma de ejecutar el programa es desde la línea de comandos tal y como muestra la Figura 8.b).

### 4.3. Utilizando ficheros de entrada

Se pueden utilizar ficheros de datos de validación como el mostrado en el fichero **v\_0inside.test** en la Sección 9. Para ello es conveniente crear una carpeta específica para almacenar estos ficheros de validación, la cual se podría llamar **tests**. Desde la línea de comandos tan solo tenemos que redirigir la entrada desde este fichero (ver Figura 9).

```

Output x
MPGeometry (Clean, Build) x MPGeometry (Build, Run) x MPGeometry (Run) x
First rectangle is [(2,5) - (5,2)]
Type second rectangle: 4 3 8 0
Calculating intersection of: [(2,5) - (5,2)] and [(4,3) - (8,0)]
The intersection is: [(4,3) - (5,2)]
Reading points...0 0 1 1 -1 0
None of them falls within the intersection
RUN FINISHED; exit value 0; real time: 33s; user: 0ms; system: 0ms

```

(a)

```

Terminal - ...uiz@mdruiz-MS-7B29: ~/MP/MPGeometry x Output
mdruiz@mdruiz-MS-7B29: ~/MP/MPGeometry$ dist/Debug/GNU-Linux/mpgeometry
First rectangle is [(2,5) - (5,2)]
Type second rectangle: 4 3 8 0
Calculating intersection of: [(2,5) - (5,2)] and [(4,3) - (8,0)]
The intersection is: [(4,3) - (5,2)]
Reading points...0 0 1 1 -1 0
None of them falls within the intersection
mdruiz@mdruiz-MS-7B29: ~/MP/MPGeometry$

```

(b)

Figura 8: Ejecutando el binario. a) Desde el menú “Run Project”. b) Desde la shell en la carpeta del proyecto

```
dist/Debug/GNU-Linux/mpgeometry < tests/v_0inside.test
```

```

Terminal - ...uiz@mdruiz-MS-7B29: ~/MP/MPGeometry x Output
mdruiz@mdruiz-MS-7B29: ~/MP/MPGeometry$ dist/Debug/GNU-Linux/mpgeometry < tests/v_0inside.test
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
mdruiz@mdruiz-MS-7B29: ~/MP/MPGeometry$

```

Figura 9: Ejecutando el binario con redireccionamiento de entrada desde la shell

Si se quiere ejecutar esta redirección de datos de entrada para validar el programa desde el menú de Netbeans, es necesario acceder a las propiedades del proyecto desde la pestaña “Projects”, botón derecho del ratón, Properties → Run → Run Command o bien desde el menú: **File - Project Properties - Run - Run Command** e introducir las modificaciones según se muestra en la Figura 10 para indicar que el binario se va a ejecutar redireccionando la entrada estándar desde **v\_0inside.test**.

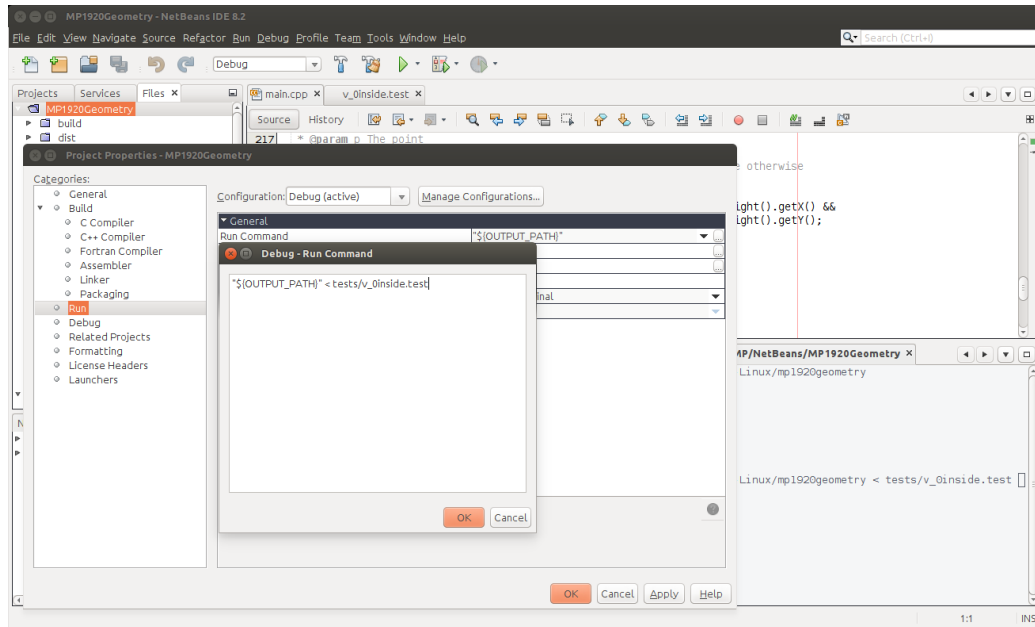


Figura 10: Ejecutando el binario con argumentos o redireccionamiento de entrada

#### 4.4. Compilación separada

En la inmensa mayoría de las veces los programas no se implementan en un único archivo, como es el caso del programa de la sección anterior. De hecho, este módulo representaría una mala praxis en programación.

Lo normal es encontrarnos la implementación separada en distintos módulos o archivos, donde la norma es incluir todos los elementos que pudiesen estar relacionados en un único módulo. 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.

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`.

#### 4.4.1. 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 utilizar para calcularla. Por ejemplo en C++11, nos encontramos con las distintas funciones

```
/** Calcula la raíz cuadrada
 * @param x número
 * @return raíz 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 sean utilizados por otros módulos) lo debemos de hacer público (debe estar incluido como tal en el `.h`).

#### 4.4.2. 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 5). 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 en adelante describiremos cómo crear un proyecto más complejo y dividirlo en múltiples ficheros con NetBeans.

## 4.5. Modularización: separando el programa en múltiples ficheros

En proyectos complejos se hace necesario dividir el programa en múltiples ficheros, de forma que cada uno de ellos soporte una parte de la implementación total. De esta forma, separaremos el programa original en los siguientes ficheros, tal y como se detalla en los listados de la Sección 10. 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 **MP-GeometryV2** donde seguiremos los siguientes pasos.

### 4.5.1. 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 11.a se pueden separar las implementaciones de los métodos fuera de la clase usando:

```
NombreDeLaClase :: nombreMetodo
```

<pre>#include &lt;iostream&gt; using namespace std;  class Helloworld { private:     string message; public:     Helloworld() {         message = "Hello_world!";     }     void print() {         cout &lt;&lt; endl &lt;&lt; message &lt;&lt; endl;     }     void set(string s) {         message = s;     } };  int main() {     Helloworld hw;     hw.set("Hola_mundo!");     hw.print();     return 0; }</pre>	<pre>#include &lt;iostream&gt; using namespace std; class Helloworld { private:     string message; public:     Helloworld();     void print() const;     void set(const string &amp;s); };  int main() {     Helloworld hw;     hw.set("Hola_mundo!");     hw.print();     return 0; }  Helloworld::Helloworld() {     message = "Hello_world!"; }  void Helloworld::print() const {     cout &lt;&lt; endl &lt;&lt; message &lt;&lt; endl; }  void Helloworld::set(const string &amp;s) {     message = s; }</pre>
--	--

Figura 11: Separar declaración/definición de implementación. Obsérvese a la derecha el uso del cualificador `Helloworld::` en las implementaciones de los métodos para indicar que se trata de métodos de esa clase, no de funciones genéricas

como puede verse en la Figura 11.

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

En el fichero de especificación (`.h`) irán las definiciones/declaraciones de los datos y los prototipos de los métodos o funciones, esto es todo lo que necesite conocer otro 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 sólo se incluirán una vez en nuestro código, es decir, evitaremos las dobles inclusiones. No incluirlo es uno de los errores mas comunes, como veremos posteriormente.

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



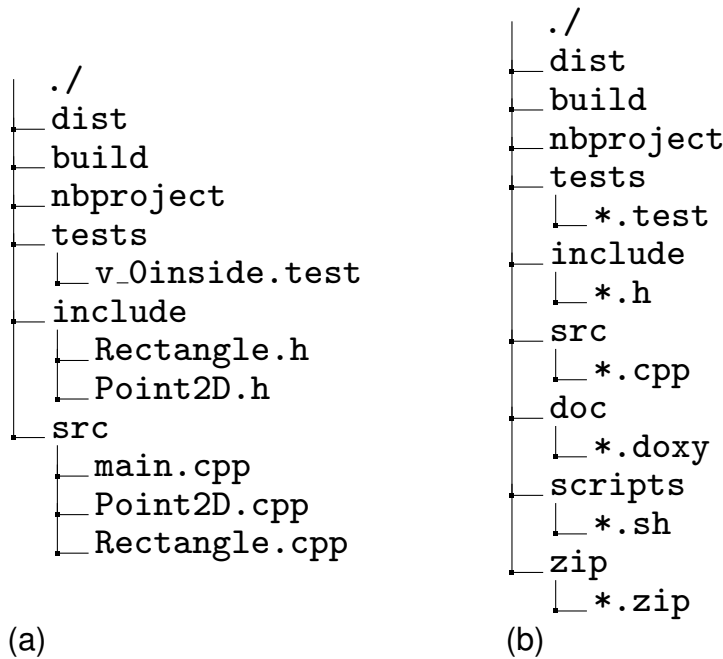


Figura 12: a) Estructura básica de las carpetas de un proyecto con múltiples módulos, Las tres primeras subcarpetas (**dist**, **build** y **nbproject**) son creadas y mantenidas automáticamente por NetBeans. El resto (**tests**, **src**, **include**) deben crearse a mano en cada proyecto. b) Estructura extendida de carpetas para la gestión de un proyecto que incluyen las carpetas **doc** (para almacenar ficheros Doxygen y documentación del proyecto), **scripts** para almacenar scripts de gestión del mismo como los que se puede descargar de Prado, y la carpeta **zip** para guardar copias de seguridad del proyecto.

Después 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 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"
```

## 4.6. Crear la estructura del proyecto en la vista física

De esta forma, será necesario organizar la carpeta interna del proyecto para que todo esté en su sitio, tal y como muestra la Figura 12.

Para crear esta estructura en NetBeans hay que hacerlo desde la pestaña "Files" del navegador de proyectos, crear la estructura que muestra la Figura 12 quedando como muestra la Figura 13.a)

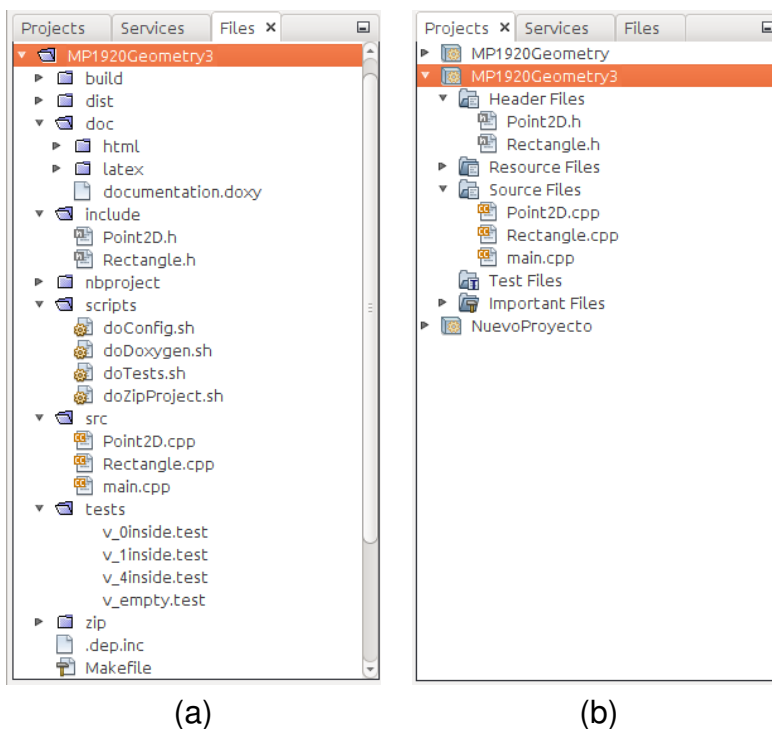


Figura 13: Estructura de carpetas del proyecto, en su visión física (a) y lógica (b)

## 4.7. Configurar la vista lógica del proyecto

En la vista lógica del proyecto, pestaña “Projects” es necesario indicar en qué carpeta se encuentran los ficheros .h y .cpp. Para ello se procede como sigue:

1. Header files. Pulsar con el botón derecho, “Add existing item...” y navegar hasta donde están los ficheros .h, seleccionarlos y añadirlos.
2. Source files. Pulsar con el botón derecho, “Add existing item...” y navegar hasta donde están los ficheros .cpp, seleccionarlos y añadirlos.

Esta operación no añade nuevos ficheros al proyecto, tan sólo le indica a NetBeans dónde están estos. Al concluir esta etapa, el proyecto aparece como muestra la Figura 13.b).

## 4.8. Configurar los parámetros del proyecto

Ya casi está terminado. Las opciones del proyecto se definen desde la pestaña de propiedades del proyecto (Figura 14), la cual aparece con botón derecho, “Properties”. En esta ventana cambiaremos los siguientes parámetros:

1. En la categoría “Build” - “C++ compiler”: En “Include directories” se selecciona la carpeta “include” perteneciente al proyecto.

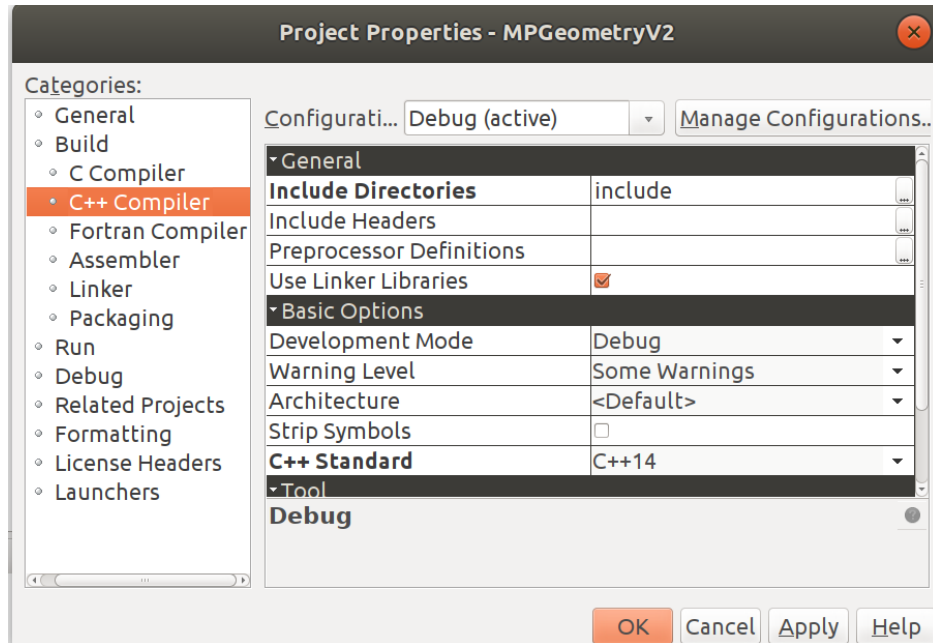


Figura 14: Propiedades del proyecto

2. En “Basic Options” - “Development Mode” podremos elegir entre “Debug”(Modo de depuración) o “Release” (Modo de producción). Para recompilar todo el proyecto se elige en el campo **Configuración** y se elige **Debug** o **Release** respectivamente.
3. En “Basic Options” - “C++ Standard”, seleccionaremos bajo qué estándar de C++ se va a compilar el proyecto. Seleccionaremos siempre “C++14”.

Para introducir parámetros en la llamada al programa o redireccionamientos de la entrada o salida del programa (por ejemplo con ficheros de validación de datos) es necesario modificar las propiedades del proyecto: En la categoría “Run” - “Run Command” hay que editar el valor actual para que quede con los valores de

```
"${OUTPUT_PATH}" < tests/v_0inside.test
```

## 5. Bibliotecas

En NetBeans cada proyecto que se crea sirve para generar un único binario, de forma que para generar más de un binario habrá que crear más de un proyecto. Igualmente pasa con las bibliotecas para las que hace falta crear un proyecto individual (Figura 15). Desde el menú **File** o desde el navegador de proyectos será necesario crear un nuevo proyecto e indicar **C/C++ Static Library**, en vez de **C/C++ Application**. Le daremos un nombre al proyecto, que será el nombre de la biblioteca **Geometry** y se gestionará como un proyecto independiente de NetBeans.

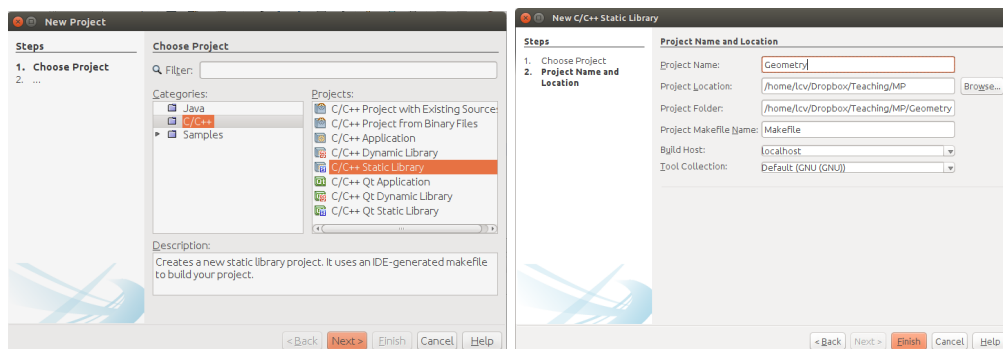


Figura 15: Creando un nuevo proyecto para gestionar una biblioteca

### 5.1. Crear la estructura de ficheros

1. En el árbol físico del proyecto de la biblioteca crear la estructura necesaria (carpetas **src** e **include** nada más) e incluir los ficheros oportunos (Figura 16.a).
2. Configurar el proyecto para definir la carpeta de ficheros de cabeceras igual que se hizo en el proyecto anterior (ver Figura 14) en el parámetro **Include Directories**.
3. En el árbol lógico del proyecto, añadir los **Header Files** y los **Source Files** que acabamos de duplicar usando el menú contextual (botón derecho del ratón) **Add Existing Item** y seleccionando los ficheros que acabamos de introducir (Figura 16.b).
4. Construir el proyecto de biblioteca (**Clean and Build Project**) igual que se hizo en el proyecto anterior, con la diferencia de que, en este caso, no se ha generado un binario, sino una biblioteca. En la Figura 17 se puede ver cómo NetBeans llama adecuadamente al compilador para compilar las fuentes y, posteriormente, al programa **ar** para generar la biblioteca. La biblioteca recién generada se encuentra en la carpeta **dist** tal y como muestra la Figura 18.a).

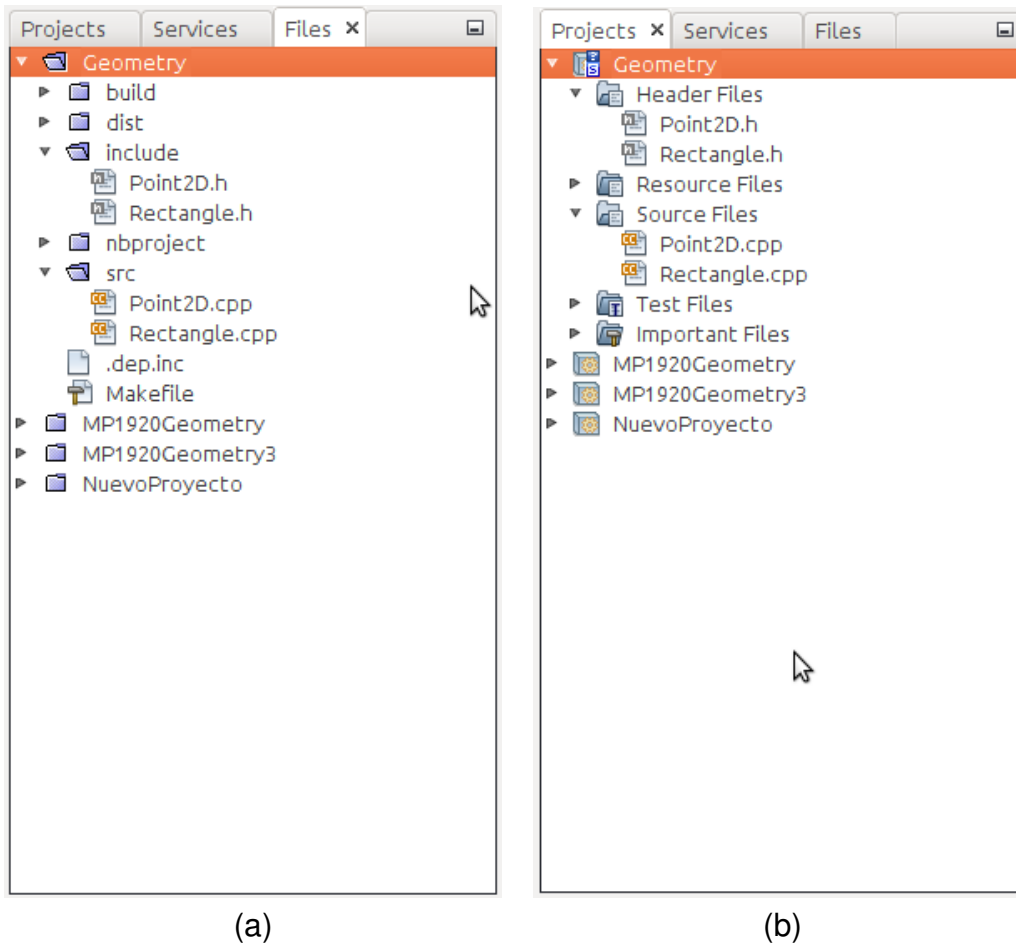
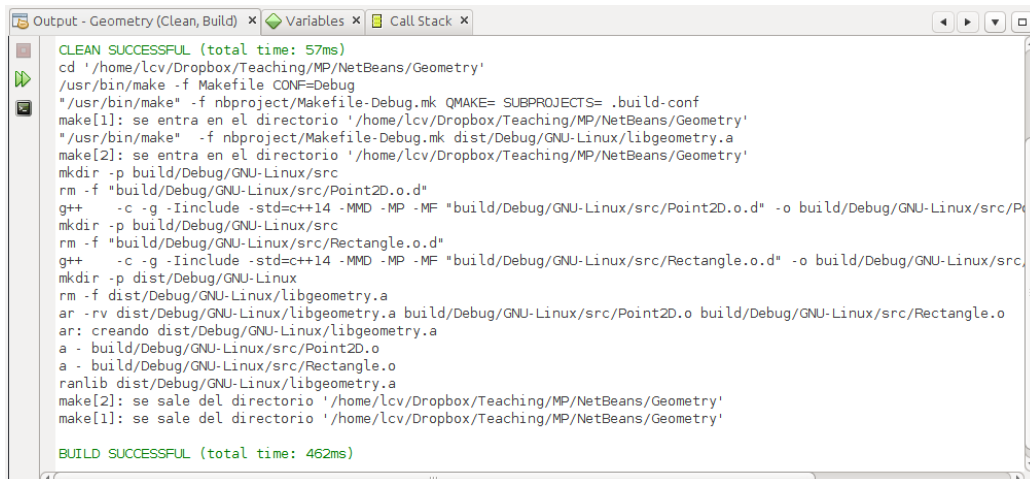


Figura 16: Vistas física (a) y lógica (b) del nuevo proyecto de biblioteca

## 5.2. Usar la nueva biblioteca en el proyecto

Siguiendo el ejemplo de la intersección de rectángulos, el nuevo proyecto, que constaría de dos proyectos vinculados entre sí, uno para la biblioteca recién creada y otro para el programa principal, que debería constar sólo del fichero **main.cpp**, el resto de ficheros desaparecen del proyecto.

1. En el árbol lógico del proyecto los ficheros .h y .cpp ya no aparecen porque han pasado a formar parte de la biblioteca y ya no se van a usar más desde aquí, sino desde el proyecto de la biblioteca. El nuevo árbol lógico queda como muestra la Figura 18.b).
2. Integración de la biblioteca en el proyecto. Esto se puede hacer de dos formas distintas.
  - a) Colocar los ficheros .h y .a de la nueva biblioteca en las carpetas que Linux tiene preparadas para esto y que son **/usr/local/include** y **/usr/local/lib** para que se puedan reutilizar de ahora en adelante por todos los proyectos futuros.
  - b) Vincular los proyectos de NetBeans del programa principal y la biblioteca. Para ello se selecciona el menú



```
Output - Geometry (Clean, Build) x Variables x Call Stack x
CLEAN SUCCESSFUL (total time: 57ms)
cd '/home/lcv/Dropbox/Teaching/MP/NetBeans/Geometry'
/usr/bin/make -f Makefile CONF=Debug
"/usr/bin/make" -f nbproject/Makefile-Debug.mk QMAKE= SUBPROJECTS= .build-conf
make[1]: se entra en el directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/Geometry'
"/usr/bin/make" -f nbproject/Makefile-Debug.mk dist/Debug/GNU-Linux/libgeometry.a
make[2]: se entra en el directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/Geometry'
mkdir -p build/Debug/GNU-Linux/src
rm -f "build/Debug/GNU-Linux/src/Point2D.o.d"
g++ -c -g -Iinclude -std=c++14 -MMD -MP -MF "build/Debug/GNU-Linux/src/Point2D.o.d" -o build/Debug/GNU-Linux/src/Point2D.o
mkdir -p build/Debug/GNU-Linux/src
rm -f "build/Debug/GNU-Linux/src/Rectangle.o.d"
g++ -c -g -Iinclude -std=c++14 -MMD -MP -MF "build/Debug/GNU-Linux/src/Rectangle.o.d" -o build/Debug/GNU-Linux/src/Rectangle.o
mkdir -p dist/Debug/GNU-Linux
rm -f dist/Debug/GNU-Linux/libgeometry.a
ar -rv dist/Debug/GNU-Linux/libgeometry.a build/Debug/GNU-Linux/src/Point2D.o build/Debug/GNU-Linux/src/Rectangle.o
ar: creando dist/Debug/GNU-Linux/libgeometry.a
a - build/Debug/GNU-Linux/src/Point2D.o
a - build/Debug/GNU-Linux/src/Rectangle.o
ranlib dist/Debug/GNU-Linux/libgeometry.a
make[2]: se sale del directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/Geometry'
make[1]: se sale del directorio '/home/lcv/Dropbox/Teaching/MP/NetBeans/Geometry'

BUILD SUCCESSFUL (total time: 462ms)
```

Figura 17: Creación de la biblioteca **libgeometry.a** de forma automática desde NetBeans

### File - Project Properties - Related Project

y se selecciona la carpeta donde está el proyecto de la biblioteca, asegurandose de marcar la casilla “Build”. De esta forma, cualquier cambio en la biblioteca haría que se recompilasen de nuevo, no solo la biblioteca, sino también el nuevo proyecto.

3. Sea cual sea la opción elegida, en las propiedades del proyecto se debe añadir la carpeta donde se encuentren los ficheros .h en la opción **Include Directories**, es decir, bien la carpeta include del sistema, bien la carpeta include del proyecto de la biblioteca. Los includes del código se pueden dejar con comillas dobles o como ángulos indiferentemente a partir de ahora.
4. En las propiedades del proyecto pero en las opciones del enlazador (**Linker** Figura 20), añadir la biblioteca recién creada en la opción **Libraries**. Elegir la ubicación de la misma igual que en el apartado anterior.
5. Crear el nuevo binario (**Clean and Build Project**).

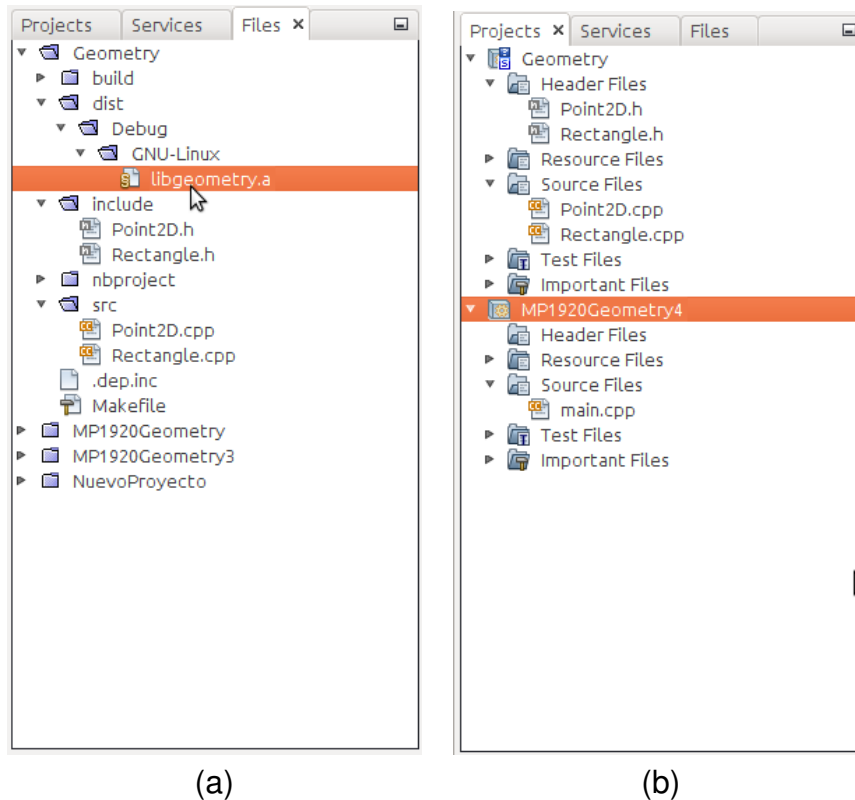


Figura 18: (a) Situación de la biblioteca **libgeometry.a** en el árbol físico del proyecto dentro de la carpeta **dist** (b) Nuevo árbol lógico del proyecto, que excluye a todos los .h y .cpp que se ya no pertenecen al proyecto principal sino a la biblioteca recién creada

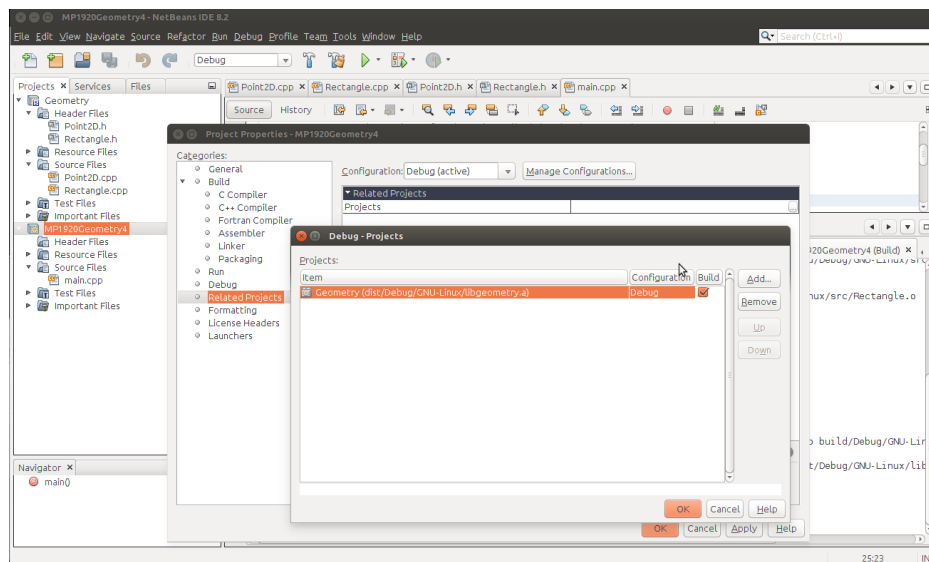


Figura 19: Configuración del proyecto para vincular el nuevo proyecto al proyecto de la biblioteca

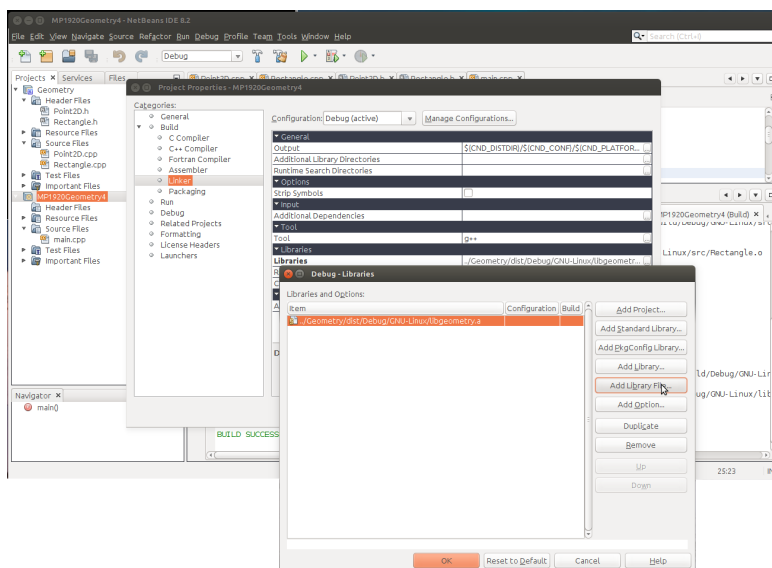


Figura 20: Incluyendo la biblioteca **libgeometry.a** para enlazarla

## 6. Depurador

### 6.1. Conceptos básicos

NetBeans actúa, además, como una interfaz separada que se puede utilizar con un depurador en línea de órdenes. En este caso será la interfaz de alto nivel del depurador **gdb**. Para poder utilizar el depurador es necesario compilar los ficheros fuente con la opción **-g** o, lo que es lo mismo, con la configuración **Debug** en la ventana de propiedades del proyecto NetBeans.

### 6.2. Ejecución de un programa paso a paso

Para comenzar a ejecutar un programa bajo control del depurador es conveniente colocar un PR<sup>5</sup>. NetBeans permite crear un Punto de Ruptura simplemente haciendo click sobre el número de línea correspondiente en la ventana de visualización de código y visualiza esta marca como una línea en rojo (Figura 21).

Una vez colocado este punto de ruptura se puede comenzar la ejecución del programa con el menú<sup>6</sup>

#### Debug - Debug Project

NetBeans señala la línea de código activa con una pequeña flecha verde a la izquierda de la línea y remarca toda la línea en color verde (Figura 21). A la misma vez, se abren una serie de pestañas adicionales en el cuadrante inferior derecho, tal y como se comentó en la Sección 2.

<sup>5</sup>Un punto de ruptura (abreviadamente PR) es una marca en una línea de código ejecutable de forma que su ejecución siempre se interrumpe antes de ejecutar esta línea, pasando el control al depurador

<sup>6</sup>Consultar ([Abrir →](#)) para una descripción más detallada de las funciones de depuración de NetBeans, tanto para C++ como para otros lenguajes.



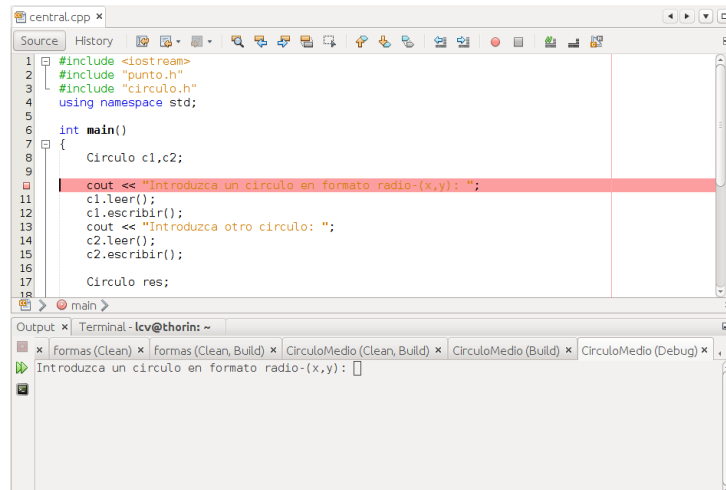


Figura 21: Creando un punto de ruptura para depurar un programa. El PR aparece como una línea de color rojo. La línea que se va a ejecutar a continuación aparece marcada en color verde

Las siguientes son algunas de las funciones más habituales de ejecución paso a paso (ver Figura 22):

- **Debug - Step Into**  
Ejecuta el programa paso a paso y entra dentro de las llamadas a funciones o métodos.
- **Debug - Step Over**  
Ejecuta el programa paso a paso sin entrar dentro de las llamadas a funciones o métodos, las cuales las resuelve en un único paso.
- **Debug - Continue**  
Ejecuta el programa hasta el siguiente punto de ruptura o el final del programa.



Figura 22: Herramientas utilizadas durante la depuración de un programa. De izquierda a derecha: Finish Debugger Session, Restart, Pause, Continue, Step Over, Step Into, Step Out, Run to Cursor

### 6.3. Inspección y modificación de datos

NetBeans, como cualquier depurador, permite inspeccionar los valores asociados a cualquier variable y modificar sus valores sin más que acceder a la pestaña **Variables** y desplegar las variables deseadas y modificarlas, en caso necesario (Figura 23).

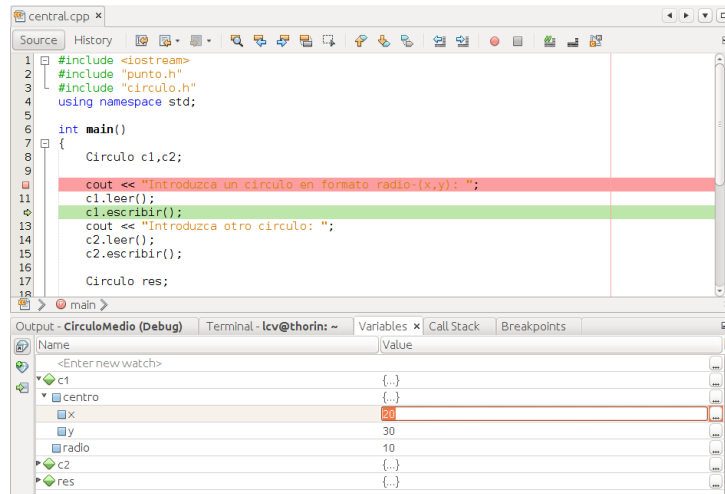


Figura 23: Inspeccionado y modificando, si fuese necesario, las variables del programa

## 6.4. Punto de ruptura condicional

Una utilidad muy interesante en Netbeans, y en cualquier depurador, es utilizar lo que se denominan puntos de ruptura condicionales que permite detener la ejecución del programa si se cumple una condición, como por ejemplo que se hayan dado 5000 iteraciones de un bucle o que una determinada variable tome un valor concreto, en el que sabemos que el programa no funciona correctamente.

En esta situación el programa se ejecutará normalmente hasta que dicha condición se satisfaga y se detiene en el lugar indicado. Para fijar el PR condicional, primero fijamos un PR normal, nos situamos sobre él y le damos al botón derecho del ratón. En el menú emergente seleccionamos la opción **Break Point - Properties**, y a continuación indicaremos la condición de parada, así como la acción **Count Limit**. Podemos ver que al punto de ruptura le ha salido una pequeña muesca que muestra que la depuración se detiene cuando se satisfaga alguna condición.

Los PR condicionales nos serán muy útiles por lo que es recomendable hacer distintas pruebas hasta sentirse seguro de su funcionamiento.

## 6.5. Inspección de la pila

Durante el proceso de ejecución de un programa se suceden llamadas a módulos que se van almacenando en la pila. NetBeans ofrece la posibilidad de inspeccionar el estado de esta pila y analizar qué llamadas se están resolviendo en un momento dado de la ejecución de un programa (Figura 25).

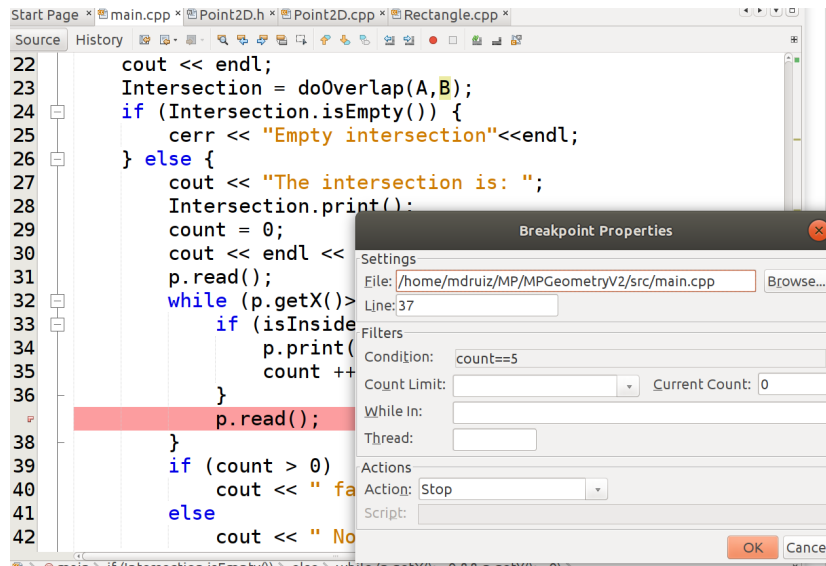


Figura 24: Punto de Ruptura condicional

## 6.6. Reparación del código

Durante una sesión de depuración es normal que sea necesario modificar el código para reparar algún error detectado. En este caso es necesario mantener bien actualizada la versión del programa que se encuentra cargada. Para ello lo mejor es interrumpir la ejecución del programa

### **Debug - Finish Debugger Session**

y re-escribir los cambios y recompilarlos (**Clean and Rebuild Project**).

## 7. Otras características de NetBeans

Además de estas características, el editor de código de NetBeans dispone de algunas ayudas a la escritura de código que incrementan enormemente la eficiencia y la detección temprana de errores de programación. Estas son algunas de estas características.

1. Ayuda a la escritura de llamadas a métodos y funciones (Figura 26). Mientras se escribe la llamada a un método se muestran las distintas posibilidades, sus parámetros y la información de ayuda.
2. Ayuda a la escritura de llamadas a métodos y funciones (Figura 27). Si se escribe una llamada a un método inexistente o con una llamada incorrecta, NetBeans lo señala inmediatamente con un icono rojo en la línea de la llamada.
3. Ayuda a la escritura de variables (Figura 28). Si se escribe una variable no declarada aún, NetBeans lo señala inmediatamente con un icono rojo en la línea de la llamada.

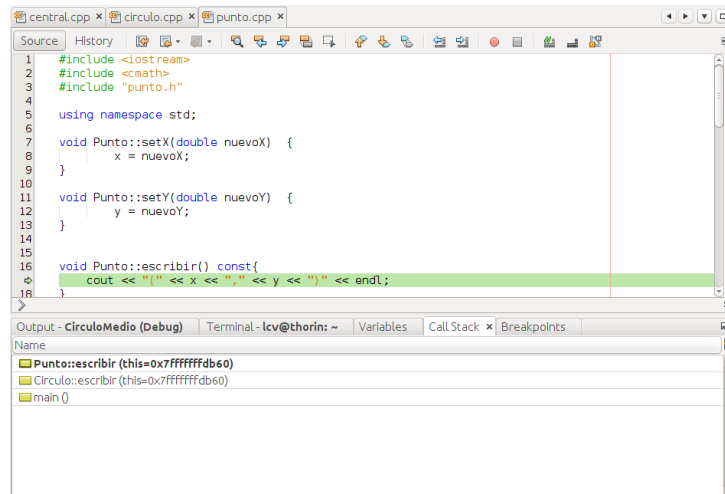


Figura 25: Inspeccionado y modificando, si fuese necesario, la pila de llamadas del programa

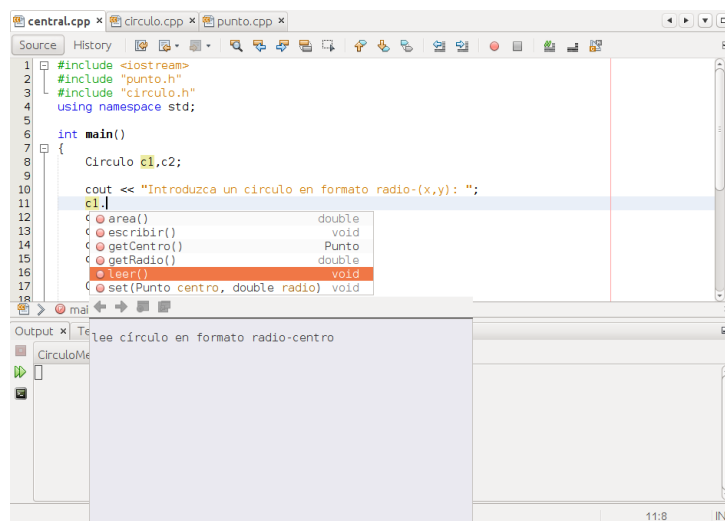


Figura 26: Auto-rellenado de código

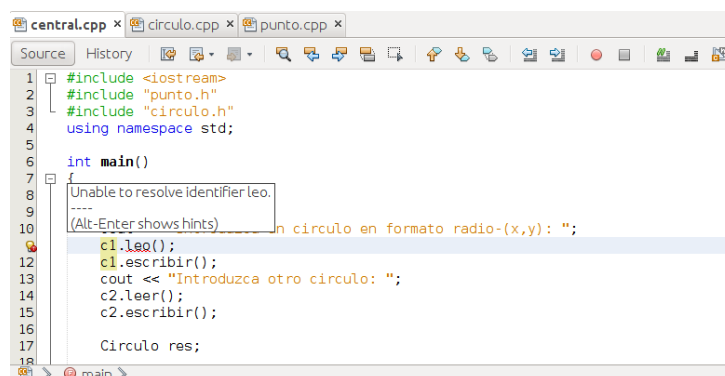


Figura 27: Detección inmediata de llamadas erróneas

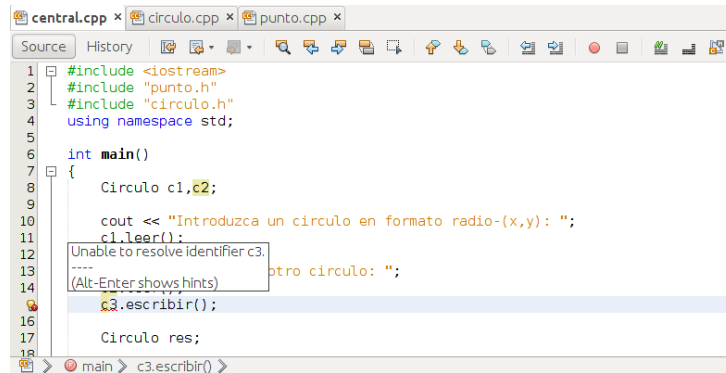


Figura 28: Detección inmediata del uso de identificadores erróneos

4. Ayuda a la indentación de código (Figura 29). Cada vez que se escribe código en una ventana, al pie de la misma aparece una indicación del nivel de anidamiento de código en el que se encuentra.

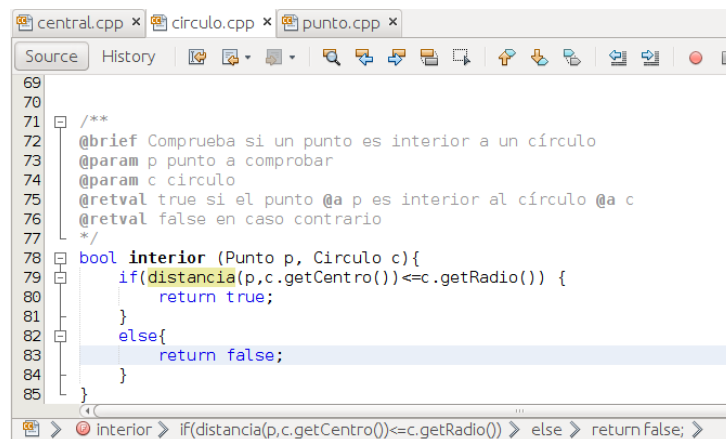


Figura 29: Detección inmediata del nivel de anidación del código. En la parte inferior del editor se puede observar la localización precisa de la línea que se está editando (línea número 27): Función principal **main**, dentro de un **if** y en la parte **else** del mismo

## 8. Personalización de NetBeans

NetBeans puede ejecutar scripts del sistema operativo (Linux) para realizar tareas externas al propio entorno, 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 Prado, 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).

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

**doDocumentation.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.

**doZipProject.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.

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

## 9. El programa original

Todos estos ficheros están disponibles en Prado para su descarga en la página de la asignatura.

```
#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;
    }
}

```

## 9.1. 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 8).

```

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

```





## 10. Apendice 2. Modularización

### 10.1. 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
```

### 10.2. 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
```

### 10.3. 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 << ")";
}
```



## 10.4. 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 << "]" << endl;
}

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();
}
```

## 10.5. 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 << "All within the intersection (" << count << " total)" << endl;
        else
            cout << "None of them falls within the intersection" << endl;
    }

    return 0;
}
```

## 11. 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 "xxx.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 { }; // Redefinición 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 si sustituimos el `include` de `.h` por su contenido.

```

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

(main.cpp)
#include "A.h"
int x;
=====
(lo que se obtiene al sustituir 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

# 12. Videotutoriales

1. Preparación del entorno de trabajo ([Abrir →](#)).
2. Proyecto **MPGeometry**, descomposición modular y compilación no separada ([Abrir →](#)).
3. Proyecto **MPGeometry** modular y su transformación en una biblioteca ([Abrir →](#)).