UNIVERSIDAD DE GRANADA

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

DECSAI

Metodología de la Programación

Curso 2025/2026

Guion de prácticas

El depurador

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

Índice

1. Conceptos básicos

La cadena de herramientas de desarrollo de software que venimos utilizando incluye: el editor de código **VSCode**, el compilador **gcc**/**g++** y el sistema de construcción **cmake**, nos falta conocer una última herramienta, el depurador. El depurador tiene como función ayudar al programador a detectar, diagnosticar y corregir los errores (bugs) en el código, su utilización reduce drásticamente el tiempo dedicado a la fase de prueba y depuración de un programa. Indicar que, esta herramienta tiene su lugar entre una vez elaborado el código fuente (o durante su elaboración) y la obtención del binario ejecutable.

VSCode como editor centrado en la **edición de código**, no incluye un depurador propio para lenguajes como C/C++. En su lugar, utiliza el **GNU Debugger** (gdb) como su motor de depuración principal a través de una de las extensiones de C++ que instalamos en la configuración del interfaz gráfico de usuario, GUI. La combinación de la GUI y GDB nos va a permitir una depuración potente y visual para el desarrollo de software.

Para que la depuración funcione, el código fuente debe haberse compilado previamente con la bandera de depuración habilitada (`-g` en *gcc/g++*). Este modo de compilación se conoce como modo **Debug**. El modo Debug, genera un binario más grande debido a que inserta marcas y símbolos necesarios para llevar a cabo la depuración del código fuente línea a línea.

Por el contrario, el modo **Release** está optimizado para el rendimiento y el uso final del ejecutable. El binario resultante por tanto, es más pequeño, rápido y está listo para ser distribuido [1]. En la GUI puede seleccionar alguno de los modos en la barra de estado, el modo por defecto es el modo Debug. Para conmutar entre ellos fíjese en la figura 1 en el la configuración activa, puede verse `Debug`, pinche en él con el ratón y aparecen las otras variantes disponibles.

2. Ejecución de un programa paso a paso

Para explorar el depurador vamos abrir el proyecto operativo `CNN` cuyo fuente completo puede encontrarse en la Sección 5.

Se trata de un programa que crea varias instancias CNN de pares *carácter-valor numérico*, usando para ello el constructor con 0, 1 y hasta 2 valores por defecto. A continuación, el programa lee el valor de un incremento por teclado que se acumula al componente numérico de cada uno de los objetos creados. En la Figura 1 pueden verse detalles del programa.

Una posible salida de la ejecución del programa en la terminal podría ser:

```
10
10
? 31740
? 31735
```

[1] Estos son los modos más conocidos. Existen otros modos, que resultan en binarios con propósitos, características y rendimientos diferentes.

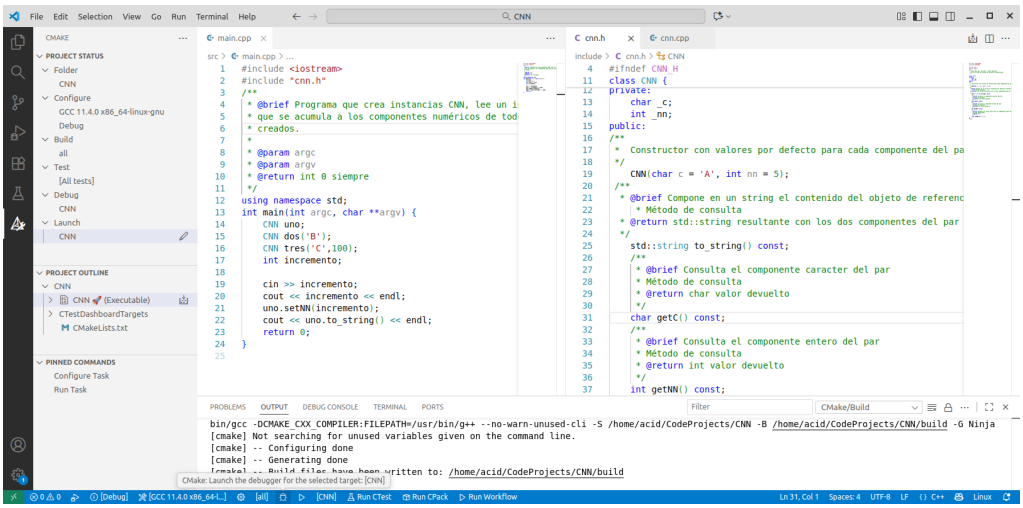

Figura 1: Punto de partida para la depuración. La clase CNN y el main.

```
5 | ? 31735
```

La línea (1) corresponde a la entrada de datos y la (2) lo leído. Líneas (3)(4) y (5) corresponden a la salida de los objetos tt uno, dos y `tres`, respectivamente.

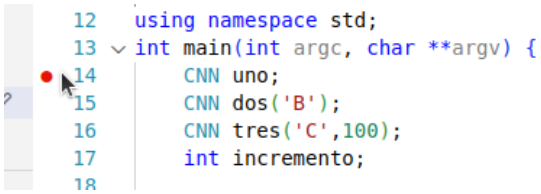

Figura 2: Creando un punto de ruptura para depurar un programa. El punto de ruptura aparece como una línea de color rojo.

Para comenzar a ejecutar un programa bajo control del depurador es conveniente colocar un punto de ruptura (breakpoint). Se trata de una marca en una línea ejecutable del código de forma que su ejecución siempre se interrumpe **antes** de ejecutar esta línea, pasando el control al depurador. La GUI permite crear un punto de ruptura simplemente haciendo click sobre el número de línea correspondiente en el editor de código y visualiza esta marca como un punto rojo, ver Figura 2.

Una vez colocado este punto de ruptura se puede comenzar la ejecución del programa en modo depuración seleccionando el icono de la barra de estado, Figura 3. Como el programa comienza con una lectura el programa está detenido hasta que se introduce un valor por en el terminal.

Figura 3: Lanzando el depurador (debugger) desde la barra de estado.

Figura 4: Barra de Control de Depuración. De izquierda a derecha: Continue, Step Over, Step Into, Step Out, Restart y Stop. Pause alterna con Continue.

VSCode señala la línea de código activa con una pequeña flecha a la izquierda de la línea y remarca toda la línea en color. A la misma vez, se muestra una **Barra de Control de Depuración**, ver detalle en Figura 4, que aparece en el cuadrante superior junto al título de los ficheros abiertos. Así mismo, se abre una serie de pestañas adicionales en el panel lateral cuando se activa la actividad `Run and Debug`.

En la barra de control de depuración, encontramos algunas de las funciones más habituales de ejecución paso a paso de cualquier depurador:

- **Debug - Continue** (Continuar)
  Ejecuta el programa hasta el siguiente punto de ruptura o hasta el final del programa.
- **Debug - Step Over** (Siguiente línea)
  Ejecuta el programa paso a paso sin entrar dentro de una llamada a función o método, la cual se resuelve en un único paso.
- **Debug - Step Into** (Paso a la función)
  Ejecuta el programa paso a paso y entra dentro de las llamadas a funciones o métodos.
- **Debug - Step Out** (Salir de la función)
  Devuelve el control a quién llamó a la función.
- **Debug - Restart** (Reiniciar)
  Interrumpe la ejecución del programa y vuelve al inicio de la depuración.
- **Debug - Stop** (Detener)
  Interrumpe definitivamente la ejecución del programa.

2.1. Inspección y modificación de datos

VSCode, 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, véase Figura 5.

Además, se pueden fijar en el panel lateral una o más variables, para su observación a lo largo de toda una ejecución, hablamos de **Watch**. Al insertar con + en la ventana nombres de variables o expresiones nos permite monitorear su valor [2] a lo largo de toda la sesión de depuración. A

[2] A diferencia de las variables locales estos son solo de consulta.

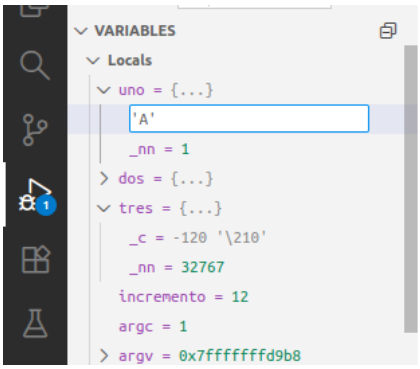

Figura 5: Inspeccionado y modificando, el campo `_c` de la variable `uno` del programa.

diferencia del panel de ”Variables Locales” que solo muestra las variables dentro del ámbito de la función, los Watches son persistentes y permiten evaluar variables o expresiones.

2.2. Inspección de la pila

Durante el proceso de ejecución de un programa se suceden llamadas a funciones que se van almacenando en la pila. VSCode ofrece la posibilidad de inspeccionar el estado de esta pila y analizar qué llamadas se están resolviendo en un momendo dado de la ejecución de un programa consultando la ventana `CALL STACK` donde figura el estado de la pila en el momento de la pausa, con los valores de los parámetros con los que fueron invocados.

2.3. 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. Interrumpir el programa, re-escribir los cambios y recompilarlos (**build**).

Ejercicio 1 *Utilizando los útiles vistos realice los cambios en el fuente para que la ejecución del programa sea el que se detalla a continuación, con un incremento de valor* 10*.*

```
10
A 15
B 15
C 110
```

Ejercicio 2 *Para adquirir las habilidades de detección de errores, dispone de un proyecto llamado Errores, donde hay varios fuentes* `main1.cpp`, `main2.cpp` *y* `main3.cpp` *con errores semánticos.*

3. Punto de ruptura condicional

Una utilidad muy interesante 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 o supere un valor concreto, se trata de recrear la situación que sabemos que el programa no funciona correctamente. En esta situación el programa se ejecutará sin interactividad hasta que dicha condición se satisfaga y se detiene en el lugar indicado.

Supongamos que se inserta el siguiente bucle antes de la salida por pantalla del contenido de los 3 objetos.

```
for (int i = 0; uno.getNN() < tres.getNN(); i++) {
    uno.setNN(uno.getNN() + incremento);
}
```

Para fijar el punto de ruptura condicional, primero nos situamos con el cursor en la línea, se pulsa botón derecho y se despliega una lista de tipos de punto de ruptura, se selecciona `Add Conditional Breakpoint...` seleccionamos `expressions` porque queremos detener la ejecución, por ejemplo, cuando el valor numérico de `uno` supere al de `dos` o bien, cuando `i == 3` que es el caso se que se muestra en la Figura 6.

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 punto de ruptura condicionales serán de gran utilidad, por lo que, es recomendable hacer distintas pruebas hasta sentirse seguro de su funcionamiento.

4. Configuración para la Depuración

Hasta ahora, hemos ejecutado y depurado un programa sin configuración explícita para la depuración. Sin embargo, para situaciones de depuración más complejas es necesario crear un archivo **launch.json** para especificar la configuración del depurador. Por ejemplo, para lanzar un programa con datos de entrada usando redireccionamiento [3] o bien para especificar parámetros a la función `main()`.

Crear un archivo de configuración de lanzamiento también es útil porque permite configurar y guardar los detalles de la configuración de depuración junto al proyecto [4].

El método más sencillo y recomendado es dejar que el asistente de VSCode genere el archivo, basándose en el lenguaje C++. Con la actividad `Run and Debug` activa, vamos a crear el archivo `launch.json`.

[3] De esta forma puede coger los datos de un fichero de entrada.

[4] VSCode almacena la información de la configuración de depuración en un archivo launch.json ubicado en la carpeta `.vscode` del espacio de trabajo (carpeta raíz del proyecto), o en la configuración de usuario.

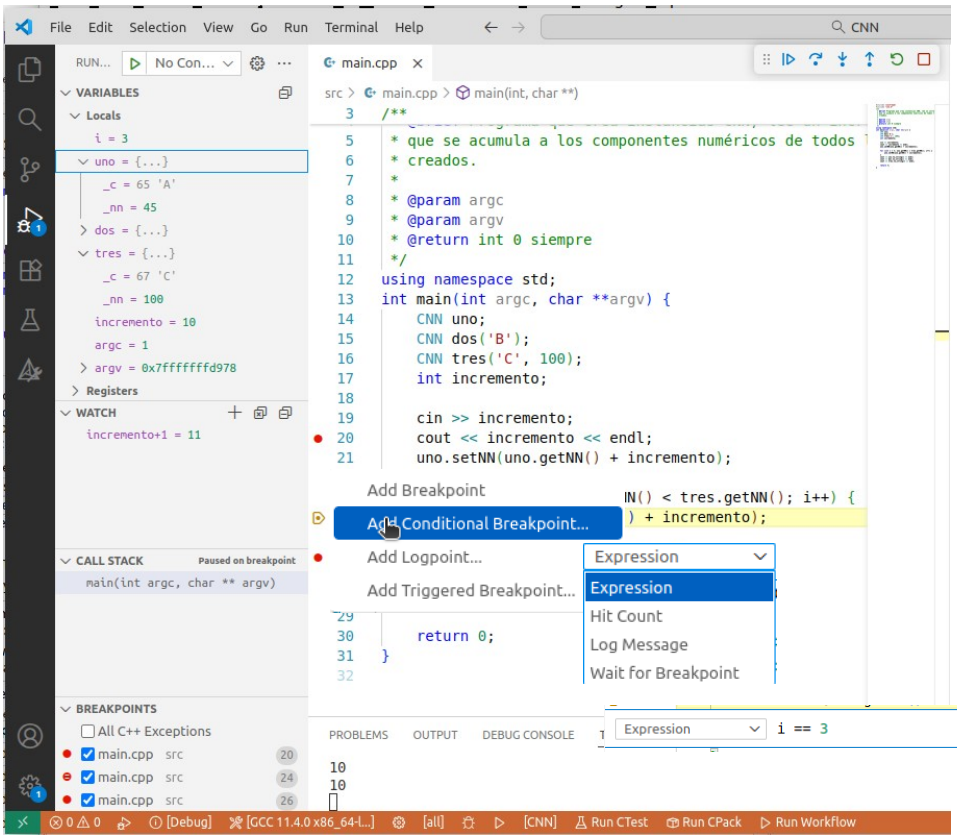

Figura 6: Punto de Ruptura condicional

Para ello, seleccione `To customize Run and Debug`. VSCode requerirá la selección del entorno (lenguaje/tipo de depurador) que se desea utilizar. En nuestro caso, `C/C++ (GDB Launch)` [5].

El resultado es un fichero `launch.json`, que contiene campos que definen parámetros y opciones de configuración para el entorno de depuración como puede verse a continuación.

4.1. Estructura Básica de `launch.json`

El archivo `launch.json` generado para C++ usando GDB podría verse así, la plantilla puede variar según la versión:

```
// Use IntelliSense to learn about possible attributes.
// Hover to view descriptions of existing attributes.
// For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
"version": "0.2.0",
"configurations": [
    {
        "name": "(gdb) Launch",
        "type": "cppdbg",
        "request": "launch",
        "program": "enter program name, for example ${workspaceFolder}/a.out",
        "args": [],
        "stopAtEntry": false,
        "cwd": "${fileDirname}",
        "environment": [],
        "externalConsole": false,
        "MIMode": "gdb",
```

[5] El modo launch es el más común y directo. Le dice al depurador que inicie (ejecute) la aplicación desde cero, desde el punto de entrada del `main()`.

```
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                },
                {
                    "description": "Set Disassembly Flavor to Intel",
                    "text": "-gdb-set disassembly-flavor intel",
                    "ignoreFailures": true
                }
            ]
        }
    ]
}
```

Indicar que, se trata de una plantilla, y **tal cual no funciona** ya que, en primer lugar hay que modificar el valor para el campo `program`, con el valor autoexplicativo de: `"enter program name, for example $workspaceFolder/a.out"`. La nueva cadena que se escribe debe contener la **ruta completa al archivo ejecutable** que GDB debe cargar para depurar. Para ello, se usa una variable de entorno `workspaceFolder` como raíz del camino.

4.2. Redireccionamiento

Sabemos ejecutar nuestro programa desde la terminal, usando datos a partir de un fichero, de modo que no se detenga en cada lectura por teclado, de la forma:

```
~usuario/CodeProjects/CNN$ ./build/CNN < tests/incremento10.txt
```

Para reproducir el redireccionamiento en la depuración, la forma más simple, es utilizar el campo `args`. Indicar que, lo que se introduce en args, simula lo que escribiría en la terminal justo después del nombre del programa, antes de presionar Enter. [6]. A continuación, se muestran los dos campos a modificar de la plantilla original para obtener una ejecución equivalente a la anterior llamada a CNN.

```
 "program": "${workspaceFolder}/build/CNN",
    "args": ["<../tests/incremento10.txt"],
```

Nota: no incluir espacios en blanco en el literal de args, en caso contrario se debería de escribir de la forma:
`"args": ["<", " ../tests/incremento10.txt"],`
pues, hay más de una cadena en args. Este campo es un array de cadenas de texto (array de cstring) que define los argumentos de la línea de comandos que se pasarán al ejecutable al iniciarse la sesión de depuración.

[6] Aunque un redireccionamiento no es un argumento propiamente dicho, como se verá después.

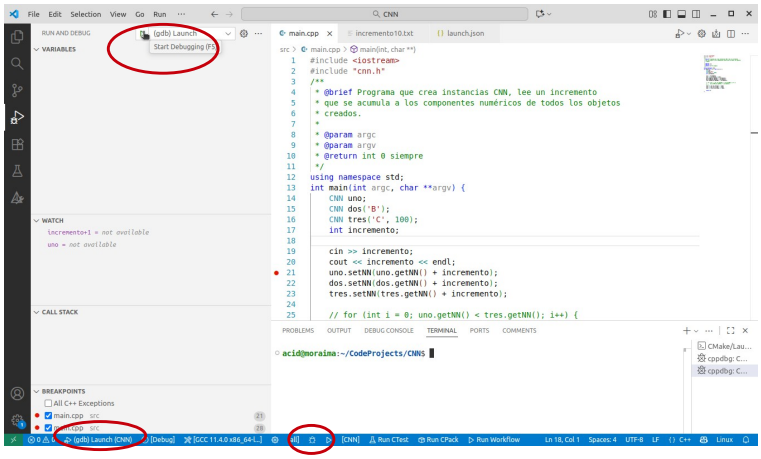

Figura 7: Distintas formas de lanzar la depuración una vez configurado GDB con `launch.json`.

4.3. Argumentos de main

El concepto que se ve aquí se desarrolla en el guion Paso de argumentos a la función main(), donde se explica como la función `main()` puede recibir parámetros desde la línea de comandos al invocarse el programa y cómo usarlos dentro de la misma. Un ejemplo de llamada de nuestro programa, CNN, con varios parámetros, es el siguiente:

```
~/CodeProjects/CNN$ ./build/CNN -k 5 localizacion
```

Podemos observar que se invoca con 3 parámetros de usuario, los cstring: `"-k"`, `"5"` y `"localizacion"`. El campo `args` del fichero `launch.json` es un **array** de **cadenas de texto** que define los argumentos de la línea de comandos que se le pasan al ejecutable al iniciarse la sesión de depuración. Luego, la línea del fichero de configuración a editar es:

```
"args": ["-k","5","localizacion"]
```

Indicar que, cada elemento definido en el array `args` se corresponde con una entrada en el array `argv` recibido por la función principal del programa: `int main(int argc, char *argv[])`. Así, con los argumentos de usuario especificados, la información que se envía al programa através de la interfaz de depuración se corresponde con:

`argc` : Tiene el valor de 4 como entero.

`argv[0]` : Contiene el nombre o ruta del ejecutable (asignado al campo `program`).

`argv[1]` : Contiene la cadena `"-k"`.

`argv[2]` : Contiene la cadena `"5"`.

`argv[3]` : Contiene la cadena `"localizacion"`.

Como podemos observar, `argc`, es el contador que controla la longitud del vector `argv`, por lo que su valor es `el número de parámetros de usuarios + 1`, por el nombre de ejecutable que se inserta en `argv[0]`. Podemos consultar las variables `argc` y `argv` en la ventana de variables locales del depurador. Y consultar con más detalle, mediante la introducción de un nuevo watch, con el valor `*argv@argc`, que nos despliega el contenido del vector `argv` con la longitud actual.

5. Proyecto CNN, código fuente y otros

5.1. cnn.h

```
#include <iostream>
#include <string>

#ifndef CNN_H
#define CNN_H
/**
 * Class CNN, par  carácter – valor numérico
 * con un mínimo de métodos de consulta y modificación.
 */

class CNN
{
private:
   char _c;
   int _nn;

public:
   /**
    *  Constructor con valores por defecto para cada componente del par
    */
   CNN(char c = 'A', int nn = 5);
   /**
    * @brief Compone en un string el contenido del objeto de referencia.
    * Método de consulta
    * @return std::string resultante con los dos componentes del par separados por espacio.
    */
   std::string to_string() const;
   /**
    * @brief Consulta el componente caracter del par
    * Método de consulta
    * @return char valor devuelto
    */
   char getC() const;
   /**
    * @brief Consulta el componente entero del par
    * Método de consulta
    * @return int valor devuelto
    */
   int getNN() const;
   /**
    * @brief Establece el nuevo valor para el componente entero del par
    * Método Modificador
    * @param valor
    */
   void setNN(int valor);
};
#endif
```

5.2. cnn.cpp

```
#include "cnn.h"
using namespace std;

CNN::CNN(char c, int n) {}

std::string CNN::to_string() const {
    std::string s = std::string(1, _c);
    s += "_" + std::to_string(_nn);
    return s;
}
char CNN::getC() const {
    return _c;
}

int CNN::getNN() const {
    return _nn;
}
void CNN::setNN(int valor){
    _nn = valor;
}
```

5.3. main.cpp

```
#include <iostream>
#include "cnn.h"
/**
 * @brief Programa que crea instancias CNN, lee un incremento
 * que se acumula a los componentes numéricos de todos los objetos
 * creados.
 *
```

```
 * @param argc
 * @param argv
 * @return int 0 siempre
 */
using namespace std;
int main(int argc, char *argv[]) {
    CNN uno;
    CNN dos('B');
    CNN tres('C', 100);
    int incremento;

    cin >> incremento;
    cout << incremento << endl;
            // se aplica el incremento al componente numérico de cada uno de los objetos
    uno.setNN(incremento);
    dos.setNN(incremento);
    tres.setNN(incremento);
    cout << uno.to_string() << endl;
    cout << dos.to_string() << endl;
    cout << tres.to_string() << endl;

    return 0;
}
```

5.4. Un fichero de datos para la evaluación

En el proyecto junto los fuentes ya mencionados, se suministra una carpeta llamada `tests/` que contiene un fichero de datos llamado `incremento10.txt`. Su contenido, se muestra a continuación. En primer lugar contiene el único dato que va a ser leído, 10. Adicionalmente de forma separada se especifica 1) cómo debería de llamarse al programa con redireccionamiento y 2) la salida en pantalla que debería de dar esa ejecución.

```
10

%%%CALL < tests/incremento10.txt
%%%OUTPUT
10
A 15
B 15
C 110
```

6. Videotutoriales

1. Uso de depurador con diferentes lenguajes (Abrir →)
2. Sesión de depuración en detalle a partir del minuto 8 (Abrir →) en `Tutorial for Beginners (For Absolute Beginners)`.