ENTORNO DE DESARROLLO DE GNU (gdb y ddd)

1. DEPURACIÓN DE CÓDIGO C EN LINUX: GDB Y DDD

1.1. El Depurador de Programas de GNU: GDB (GNU DeBugger)

GNU DeBugger o GDB es el programa depurador estándar de GNU, disponible tanto para Linux como para Windows. Este programa ofrece una potente interfaz por línea de comandos para la depuración de programas escritos en C.

En la sección 3.2 de este apartado estudiaremos una herramienta que se utiliza como interfaz gráfica para GDB. No obstante, es importante que se comprendan los conceptos básicos de depuración de código vía línea de comandos.

La interfaz de GDB se invoca llamando al comando *gdb* seguido del nombre del ejecutable a depurar:

```
$ gdb ./ejecutable
```

Entonces aparecerá la línea de comandos de *gdb*:

```
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-mingw32".

(gdb)
```

De la que se puede salir introduciendo el comando *quit* o simplemente *q*.

Es importante observar aquí que, para que se pueda depurar cualquier programa y se pueda seguir la ejecución en el código fuente, éste debe haberse compilado indicándole al compilador que incluya información de depuración. En el caso específico de *gcc*, esto se hará utilizando la opción *–g*. Así, por ejemplo, para compilar el proyecto Básico1 con información de depuración se haría:

```
$ gcc –g –obasico1 main.c
```

Y ahora se podría invocar al depurador gdb usando el comando:

```
$ gdb ./basico1
```

Veamos cómo comenzaríamos a realizar una prueba básica del proyecto básico1, que no posee errores. Podemos listar el código que hay cargado en el depurador por medio del comando **list** o simplemente **l**.

Para ejecutar el código cargado desde el principio al final (o hasta que se produzca un fallo) se utiliza el comando **run**. Sin embargo, la utilidad de este comando estriba fundamentalmente en la iniciación del código más que en la ejecución completa de todo

el programa. De hecho, en general interesará establecer puntos de ruptura o interrupción de la ejecución (*breakpoints*) en ciertas líneas del código o en ciertas funciones.

Para los efectos de una primer acercamiento a gdb, estableceremos un punto de interrupción en la función main del proyecto Basico1. Esto se hace con el comando "**break**" o simplemente "**b**" seguido del número de línea o del nombre de la función en donde la ejecución se detendrá. Por ejemplo, *break main* servirá para detener la ejecución al ser llamada la función main, en su primera línea ejecutable.

Para avanzar con la ejecución tenemos varias opciones: ejecutar la siguiente línea, por medio del comando **next** o simplemente **n**, o continuar la ejecución hasta el siguiente punto de interrupción o el final, por medio del comando **continue** o simplemente **c**. Téngase en cuenta que si la siguiente línea fuera una función el comando next la ejecutará pero no se mostrará la ejecución línea a línea de la función correspondiente (si esto se quisiese, el comando a utilizar sería **step**, o simplemente **s**).

Veamos a continuación una traza de ejecución del ejecutable básico1:

```
$ gdb ./basico1

GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-mingw32"...

(gdb) list

1     /* Created by Anjuta version 1.2.4a */
2     /* This file will not be overwritten */
3
4     #include <stdio.h>
5     int main()
6     {
7           printf("Hello world\n");
8           return (0);
9     }

(gdb) break main

Breakpoint 1 at 0x401315: file main.c, line 6.

(gdb) run

Starting program: /home/usuario/Projects/Basico1/src/basico1


Breakpoint 1, main () at main.c:6
6     {

(gdb) n

7           printf("Hello world\n");

(gdb) n
```

```
8           return (0);
(gdb) c
Continuing.


Program exited normally.
```

Veamos ahora, la depuración de un código con errores. En primer lugar, se editará en un nuevo fichero, al que llamaremos *divide.c* e incluiremos el siguiente código fuente, que habrá que compilar:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct _res_div {
      int cociente;
      int resto;
} RESULTADO_DIVISION;

RESULTADO_DIVISION divide(int dividendo, int divisor) {
      RESULTADO_DIVISION resultado;

      resultado.cociente= dividendo/divisor;
      resultado.resto= dividendo%divisor;

      return resultado;
}

int main (int argc, char* argv[]) {
      RESULTADO_DIVISION resultado;

      int dividendo, divisor;

      if (argc!=3) {
            fprintf(stderr,"Uso: %s dividendo divisor\n", argv[0]);
            return -1;
      }

      dividendo=atoi(argv[1]);
      divisor=atoi(argv[2]);

      resultado=divide(dividendo, divisor);

      fprintf (stdout,"COCIENTE: %s\n", resultado.cociente);
      fprintf (stdout,"RESTO: %s\n", resultado.resto);

      return 0;
}
```

Para poder introducir argumentos que se le pasen a la función main de entrada de todo ejecutable generado por medio de un código C, hay que ponerlos a continuación de la

palabra clave run. Esto es, en el caso anterior, si quisiéramos dividir 5 entre 2, habría que ejecutar el siguiente comando:

```
(gdb)  run 5 2
```

Sin embargo, si ejecutáis este comando con el programa anterior, comprobaréis que la ejecución del código es errónea ya que se genera un "fallo de segmentación" (segmentation fault):

```
Program received signal SIGSEGV, Segmentation fault.
```

Ahora, con la depuración averiguaremos cómo podemos detectar dónde está el error. Lo primero que haremos, entonces, será establecer un punto de interrupción en la función main:

```
(gdb) break main
```

Ejecutamos entonces el programa, con los parámetros anteriores, y ejecutamos paso a paso cada instrucción hasta llegar a la línea de código siguiente:

resultado=divide(dividendo, divisor);

En este caso, entraremos dentro de la *función divide* por medio del comando "step" o "s" y examinaremos el valor de la variable resultado, antes de ser devuelta por la función divide (esto es, antes de ejecutar la instrucción *return resultado*). Para ello, vamos a utilizar un nuevo comando: **print**, que se invoca junto al nombre de la variable cuyo valor se quiere mostrar. Por ejemplo: *print resultado*.

```
16          return resultado;
(gdb) print resultado
$1 = {cociente = 2, resto = 1}
```

Seguimos ejecutando el código de la función, hasta llegar a la llave de retorno y de ahí de vuelta al código de *main*.

Una vez alcanzamos la línea :

```
17    }
(gdb) n
main (argc=3, argv=0x3d1950) at divide.c:35
35          fprintf (stdout,"COCIENTE: %s\n", resultado.cociente);
(gdb) n
```

Descubriremos que si pulsamos nuevamente el comando next, recibimos el error. Por lo tanto, se puede deducir que es la línea 35 la que provoca el fallo. Efectivamente, el problema es que resultado.cociente es un número, pero el formato de fprintf espera una cadena (%s), por lo que habría que sustituir %s por %d.

Si ahora se vuelve a ejecutar el programa pasando como segundo argumento 0. Se produce una división por cero y por tanto un error en la ejecución. Se puede depurar sólo hasta la propia división. Una posible solución podría ser la comprobación de los argumentos dentro de la función, si uno de ellos es cero se imprime un mensaje por pantalla y se devuelve un código de error. El mensaje es necesario siempre y cuando la

salida de la función siga siendo numérica ya que un código numérico de error puede ser confundido con el resultado de una operación. Una solución más real pasa por devolver el resultado a través de un puntero, en este caso NULL si sería un código de error válido.

1.2.Una Interfaz Gráfica de Depuración: DDD (Data Display Debugger)

Hasta ahora hemos visto la depuración de código utilizando una interfaz por línea de comandos. Sin embargo, también existe la posibilidad de usar una interfaz gráfica. En concreto, en este apartado se introduce el uso de DDD, que puede ser utilizado como interfaz gráfica para GDB entre otros depuradores.

La Figura 1 muestra un ejemplo de pantalla DDD. Para acceder a esta pantalla simplemente tienes que teclear desde cualquier consola:

```
$ ./ddd
```

La tarea de abrir un código a depurar se realiza a través del submenú “Open Program” dentro del menú “File”, o simplemente pulsando la combinación de teclas Ctrl+O. Esto se muestra en las Figuras 1 y 2. Este menú, lleva a una ventana de exploración de archivos, donde puede seleccionarse el archivo a depurar. Aquí es importante, no obstante, observar que el archivo se selecciona en la columna más a la derecha, mientras que en la izquierda se muestran los directorios disponibles.

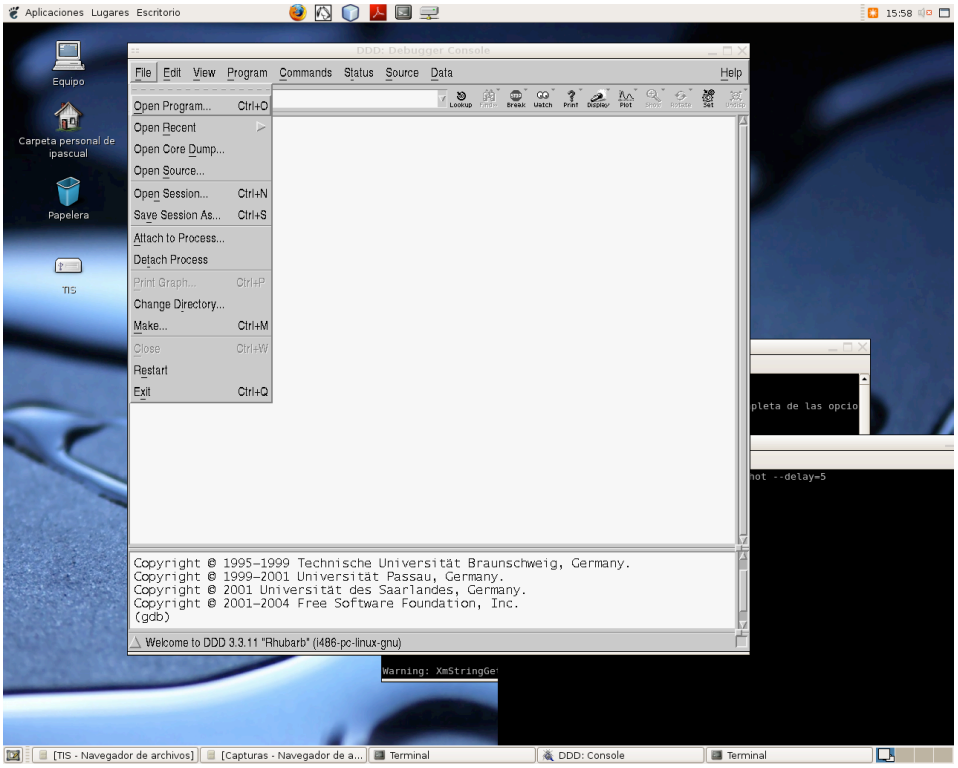

***Figura 1**. Depuración de código con ddd*

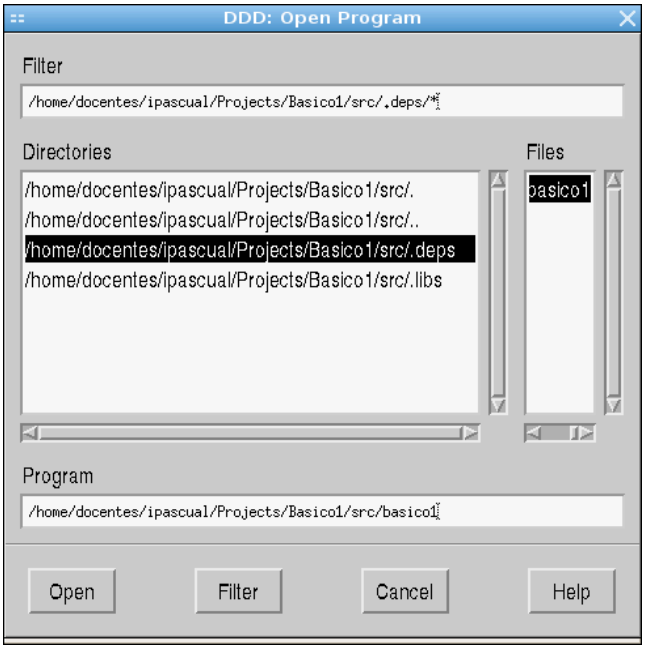

***Figura 2**. Depuración de código con ddd*

Una vez seleccionado el archivo y abierto, siempre que haya sido compilado con la opción –g de gcc, igual que en gdb (que es el depurador que se ejecuta por "debajo"), se llegará a una pantalla similar a la Figura 3.

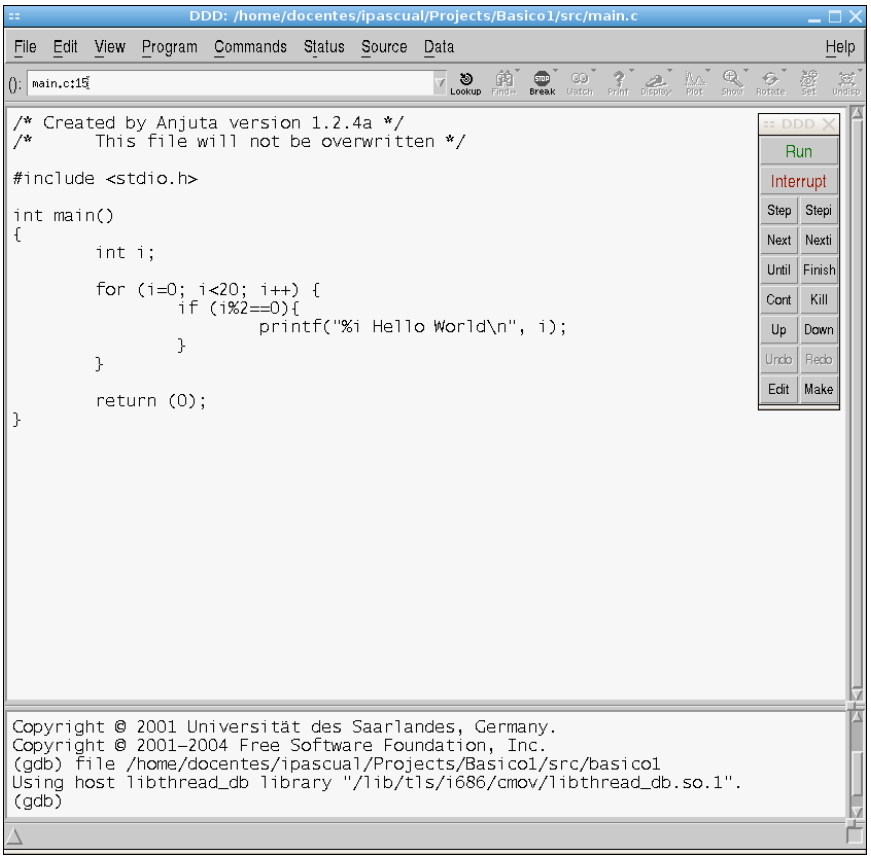

***Figura 3**. Apertura de un código a depurar con ddd*

En dicha Figura puede observarse una caja de herramientas situada por defecto en la parte superior derecha. En ella, cada botón se corresponde con un comando de los estudiados previamente en el apartado anterior. Esto es, por ejemplo, pulsar el botón RUN es equivalente a escribir run. De hecho, si se pulsa, se puede comprobar cómo, en la consola inferior de DDD aparece dicho comando (y todos los que se ejecutan como consecuencia de una acción de la interfaz).