1. Programación: el depurador gdb

Los errores (bugs) son tan inevitables como la muerte o los impuestos y el complemento perfecto de GNU gcc es GNU gdb, para el proceso de debugging. La depuración de software es otro de las etapas esenciales a la hora de crear un programa. Si lo recuerdas, era el proceso para poder identificar y corregir bugs presentes en tu código fuente. No se trata de solventar problemas de sintaxis u otro tipo de errores al escribir el código, sino de errores lógicos.

Es decir, cuando creas o escribes un código fuente y este tiene errores de sintaxis (son los que olvidas declarar alguna variable, se te olvida el punto y coma, etc., de todos estos, es el propio compilador el que revisa y te avisa en que y donde tienes los errores, las advertencias) la compilación fallará. Una vez consigues pasar esa etapa, la siguiente sería la depuración. A pesar de que el compilador haya creado el binario, eso no quiere decir que tu software esté libre de errores. Es posible que se produzcan problemas que solo se den con ciertos eventos puntuales, desbordamientos, etc. o simplemente, no hace lo que tu creíste que debía hacer. Para todo eso, te ayudará el depurador...

El depurador de GNU se llama gdb (Gnu DeBugger) y también ha sido creado inicialmente por Richard Stallman para *nix. Al igual que gcc, a no ser que se instale algún front-end, no dispone de una GUI. Pero si prefieres trabajar en modo gráfico, puedes usar *GDBtk/Insight (https://sourceware.org/insight/screenshots.php) , DDD (http://www.gnu.org/software/ddd/), etc., o IDEs como KDevelop(https://www.kdevelop.org/)*, etc. En este ejercicio, se usara emacs.

Puedes complementar gdb con herramientas como strace, Valgrind, etc.

Para depurar, hay que utilizar las opciones de *-g* y *-ggdb* cuando se usa gcc. La opción *-g* admite tres niveles, 1, 2 o 3 que especifican cuánta información de depuración incluir. El nivel predeterminado es *2 (-g2)*, que incluye gran cantidad de tablas de símbolos, números de líneas e información sobre variables locales y externas. La información para depuración de nivel 3 incluye toda la información del nivel 2 y todas las definiciones de macros presentes. El nivel 1 genera sólo la información suficiente para crear rastreos hacia atras y volcados de pila. No genera información de depuración para variables locales ni para números de línea.

1.0.1. Ejemplo 1.

Usando el código anteriormente usado, generar el código para emplear el gdb, de la siguiente manera:

```
$ gcc -g hola.c -o hola_gdb
```

No usar dicha opción va a producir que gdb nos diga en su salida que no encuentra los símbolos de depuración...

Los tamaños obtenidos cuando se compila el mismo código fuente con las opciones de *-g y -ggdb* pueden llegar a sorprender.

```
$ gcc -ggdb hola.c -o hola_ggdb
```

Se supone que no se ha eliminado el archivo binario generado anteriormente, por lo tanto, vamos a compararlo el hola con el nuevo hola_gdb. Escribimos en la terminal, en la carpeta donde estamos desarrollando los codigos, lo siguiente:

```
$ ls -l hola*
```

de esta manera podremos comparar los tres archivos el tamaño creado.

1.1. Usando GDB con Emacs

Para comenzar simplemente iniciando Emacs:

```
$ emacs
```

En el minibúfer, escriba el comando

```
M-x gdb
```

Presione la tecla de retorno después de lo cual el minibúfer mostrará cómo iniciar gdb, que es más o menos exactamente lo que escribiría en la línea de comando:

```
gdb -i = mi ./ls
```

(Una cosa útil para saber aquí es que -i=mi le dice a GDB que se está ejecutando dentro de un IDE) Presiona return en el comando en el minibúfer para iniciar GDB dentro de Emacs.

Emacs ejecutando GDB

Cuando se ejecuta GDB dentro de Emacs, el entorno IDE cambia para adaptarse a Emacs. Por ejemplo, la barra de herramientas del menú ha cambiado para mostrar los comandos de pila siguiente / anterior y arriba / abajo

Emacs también adapta la barra de herramientas a medida que la usa, agregando y eliminando opciones para ayudar a su experiencia de depuración.

1.1.1. Múltiples ventanas con GDB en Emacs

Si se configura un punto de interrupción en main y ejecutas GDB, Emacs abre automáticamente el archivo en el punto de interrupción para ti. Luego puede usar la barra de herramientas, que Emacs ha personalizado, para navegar por el programa.

Otra característica automática útil es que si continúo el programa hasta el final, la salida se muestra en una nueva ventana de Emacs para entrada / salida.

Esto es realmente útil: Emacs separa el código fuente, la salida del programa, el intérprete en ventanas separadas para que pueda navegar por su programa más fácilmente. Sin embargo, hay aún más que Emacs puede mostrar en ventanas separadas navegando al menú GUD y luego a la opción GDB-Windows (Figura 1.1.1).

Todas estas vistas en el proceso GDB son realmente útiles, pero puede comenzar a verse desordenada a medida que abre cada ventana por separado. Tiendo a no hacer esto porque hay una mejor manera de administrar todas estas ventanas que se llama gdb-many-windows .

Para usar esto, abra el minibúfer y ejecute:

```
M-x gdb-many-windows
```

Después de esto, se le muestra un diseño mucho más organizado y ordenado que parece algo que esperaría de un IDE (Figura 1.1.1)

Esto proporciona registros locales, entrada / salida, el código fuente y el comando GDB que es mucho más útil que abrir cada ventana individualmente.

Optimizando su configuración GDB / Emacs Todas estas técnicas son útiles, pero todavía hay un problema con esta configuración: no persiste en varias sesiones de Emacs, por lo que si abandono y vuelvo a Emacs, mi configuración se pierde, lo cual es realmente frustrante.

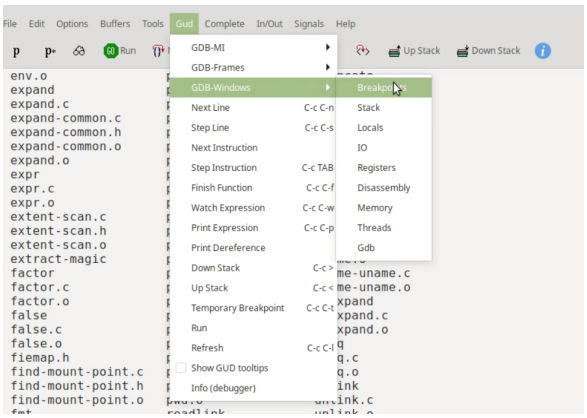

Figura 1: Menu GDB Breakpoints

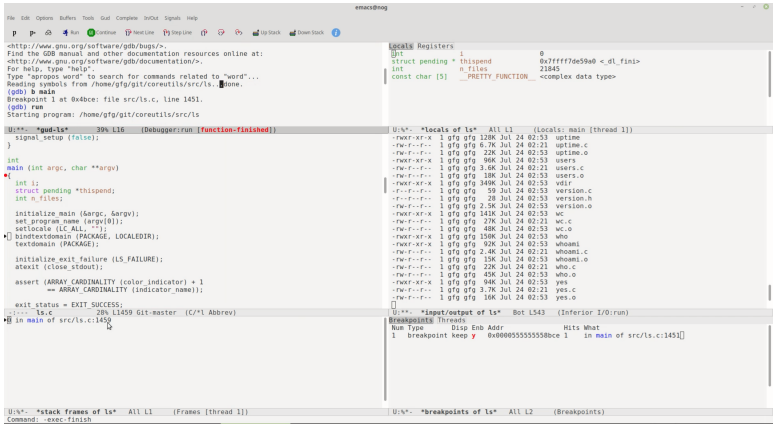

Figura 2: GDB Emacs Multiwindows

Figura 3: M-x customize

Para hacer que GDB persistan muchas ventanas, puede usar M-x customize . Esto abre las opciones de configuración de Emacs que puede buscar GDB (Figura 1.1.1)

Luego encontrará la opción de muchas ventanas y puede activarla y desactivarla. También encontrará algunas otras opciones interesantes aquí, como GDB Show main, que muestra el archivo fuente que contiene la rutina principal del inicio.

Una vez que haya hecho eso, salga de Emacs por completo, reinícielo e inicie GDB nuevamente, luego todas las ventanas útiles se han restablecido. El indicador GDB, el archivo fuente, los registros locales, locales y puntos de interrupción.

1.1.2. Establecer puntos de interrupción en GDB

Como consejo final, puede establecer un punto de interrupción GDB en Emacs haciendo clic en la canaleta junto a la línea en la que desea establecer un punto de interrupción. Cuando haces aparece un punto rojo.

1.1.3. GDB y Emacs

Lo que ves en el video es casi exactamente cómo tengo la configuración de GDB en Emacs para la depuración. Espero que esto haya sido útil y nos vemos de nuevo para el próximo gdbWatchPoint.

Un ejemplo práctico de uso

(se puede hacer tambien desde terminal en caso de que no se desee usar emacs), obtener ayuda y salir con:

```
$ gdb <-- ejecutar en terminal este comando (si usas emacs usar M-x gdb-many-windows)
(gdb) help
(gdb) quit
```

Cuando entras en el depurador, te aparece un nuevo *prompt (gdb)* mientras estés dentro para interactuar con él, como has podido comprobar.

Realizaremos el siguiente ejercicio, escribe el código de aquí que encadena unos datos introducidos para formar una línea al final con dicha información. Es muy sencillo para que comprendas bien el funcionamiento de gdb:
#include <stdio.h> #include <stdlib.h> int main() { char Saludo[]="Hola Estudiante"; char Nombre [50]; int Edad; printf("Introduce tu nombre: "); fgets(Nombre, 27, stdin); printf("Introduce tu edad: "); scanf("%d", &Edad); printf("%s%s de%d de Edad \n", Saludo,Nombre,Edad); return 0; }

Este código tiene la sintaxis correcta Y NO TIENE PROBLEMAS, por lo que gcc lo compilará sin mostrar mensaje de alerta. Pero quiero usarlo más que nada para mostrarte cómo manejarte dentro de gdb. Pudes compilarlo y seguir los ejemplos que muestro.

Ya sabes que gdb usará tanto el fichero binario como el fichero fuente de tu programa. Por eso es importante que ambos estén en el mismo directorio.

Lo deberías compilar y luego usar gdb para depurarlo:

```
$ gcc cadena.c -o cadena -g
$ gdb cadena
```

Ahora puedes usar por ejemplo list para que muestre las líneas de código de 10 en 10:

```
(gdb) list
```

También puedes pedirle que muestre una línea concreta (y unas cuantas que le preceden y otras cuantas que le siguen a la que hayas seleccionado):

```
(gdb) list 4
```

Si quieres comenzar a analizar el código para depurarlo, puedes usar:

```
(gdb) start
```

Si te fijas, ha pasado por alto todas las primeras líneas, es decir, ha comenzado a partir de la primera llave en adelante. Si quieres continuar, puedes usar next para que avance:

```
(gdb) next
```

Recuerda que puedes usar también la opciones abrevadas. Por ejemplo, en vez de next, puedes usar simplemente n

Y así hasta que quieras. Te irán apareciendo líneas y también te aparecerán las opciones interactivas del programa. Por ejemplo, en este caso, si sigues haciendo next, llegarás al mensaje que te pide tu edad, tu nombre, etc., y los podrás ir introduciendo desde gdb para analizar su comportamiento.

Si al final muestra un mensaje de de error sobre libc o similar, no te preocupes, es normal. Eso no indica nada, puedes pasarlo por alto.

Puedes usar también step para ir paso a paso, y si lo ejecutas tras una función, como por ejemplo, tras el primer printf, te puede lanzar un mensaje porque no encuentra dicha función. Eso también es normal, puesto que no se encuentra en el código fuente, sino que se encuentra en una biblioteca, en este caso en stdio.h.

```
(gdb) start
(gdb) n
(gdb) n
. . .
(gdb) step
(gdb) continue
```

Imagina que quieres crear un punto de ruptura en la línea 10, por ejemplo, es decir, en la del primer printf. Si luego usas run, arrancará hasta ese punto de ruptura, en vez de ir paso a paso. Por ejemplo:

```
(gdb) break
(gdb) run
```

Para borrar el punto de ruptura puedes usar delete:

```
(gdb) delete
```

Incluso puedes cambiar el valor de las variables. Por ejemplo, la variable Nombre, lo podemos alterar aunque ya hayamos añadido otro anteriormente. Para eso:

```
(gdb) set Nombre="hola"
(gdb) print Nombre
```

Si quieres obtener datos de una variable, como su longitud, o su tipo, puedes usar también:

```
(gdb) print strlen(Nombre)
(gdb) ptype Nombre
```

Puedes poner alertas para que cuando se modifique el valor de una variable nos muestre un aviso:

```
(gdb) watch Nombre
(gdb) info watch
```

Si en un momento dado quieres trabajar con el shell y "salir" momentáneamente de gdb para ejecutar cualquier comando, puedes usar, por ejemplo:

```
(gdb) shell ls
(gdb) shell cd ..
```

Y también crear lotes y ejecutarlo invocando el nombre que le hayamos dado como se muestra al final:

```
(gdb) define ejemplo_lote
>shell ls
>shell cat hola.c
>list
>run
>end
ejemplo_lote
```

Es un poco complicado mostrar esto así, en mi curso de Linux lo muestro en un vídeo de forma bastante más intuitiva... Pero bueno, he intentado hacerlo lo mejor posible. De todas formas, recuerda que solo quiero que aprendas a manejarte dentro de gdb y próximamente veremos más ejemplos en esta serie de posts.

Un ejemplo práctico de depuración Ahora que ya sabes moverte por gdb, puedes hacer tu primera depuración sencilla. Por ejemplo, un caso de código fuente que aparentemente está correcto, que pasaría el escrutinio de gcc, pero que presenta un error al mostrar factoriales erróneos. Aquí es donde verás mejor la utilidad de la depuración.

El código factorial.c es:

```
#include <stdio.h>
int main(void)
{
int n,
factorial, i;
do {
printf("Ingrese un número: ");
scanf("%d", &n); }
while(n<0);
for(i=1; i<=n; i++){
factorial=factorial*i; }
printf("%d! = %d", n, factorial);
```

```
return 0;
}
```

Un simple programa que pedirá un número e irá haciendo una cadena de multiplicaciones hasta conseguir el factorial. Es decir, si le ponemos 5, irá multiplicando 1 por 2, por 3, por 4, por 5, y debería dar 120. Si lo compilas y lo ejecutas, verás que el resultado que da no es el adecuado:

```
$ gcc -o factorial factorial.c -g
$ ./factorial
```

¡Algo le pasa y gcc no se ha enterado! Pero para eso está el depurador... Y aquí están los pasos:

```
gdb factorial
(gdb)
```

Vamos paso a paso, para no complicar mucho todo, voy a ir poniendo todos los pasos con comentarios diciendo qué hace cada comando (te aconsejo que lo vayas haciendo en tu sistema con el fichero fuente abierto justo en la ventana de al lado para que puedas ver el código completo y el resultado de gdb):

```
#Ejecutamos
(gdb )run
#Crea un punto de ruptura en la línea 4
(gdb) b 4
#Sigue como has aprendido con next hasta que te pida el valor de n
...
#Muestra el valor de la variable n y comprueba que vale lo mismo que tú has introducido y no ha variado
(gdb) print n
#Haz lo mismo con la otra variable factorial
(gdb) print factorial
#Como puedes ver, el valor de n es correcto, pero el de factorial es un disparate. ¡Aquí está el problema
#Puedes salir de gdb y arreglar el problema
(gdb) quit
```

Así puedes ir comprobando el valor que toman las variables en cada paso del programa y determinar si es causa de alguna de ellas, como en este caso. ¿Cómo arreglarías este desperfecto? Muy sencillo, para que factorial no se dispare al inicio, debes iniciar factorial=1, para que esté en un valor conocido y genere el resultado correcto:

```
#include <stdio.h>
int main(void) {
int n, factorial=1, i;
do {
printf("Ingrese un número: ");
scanf("%d", &n);
}
while(n<0);
for(i=1; i<=n; i++){
factorial=factorial*i;
```

```
}
printf("%d! = %d", n, factorial);
return 0;
}
```

Ahora compila nuevamente y obtendrás el binario que debe calcular correctamente:
borramamos el anterior

```
$ rm factorial
```

Compilamos el nuevo

```
$ gcc -o factorial factorial.c ./factorial
```

2. Practica

Ejecutar el programa siguiente programa.

```
#include <stdio.h>
#include <ctype.h>
int main(int argc, char **argv)
{
char c;
c = fgetc(stdin);
while(c != EOF){
if(isalnum(c))
printf("%c", c);
else
c = fgetc(stdin);
}
return 1;
}
```

localiza el problema para evitar el ciclo infinito (si te sale un ciclo infinito solo debes presionar ctrl-c).

Espero sus videos!!

Referencias

[1] Programacion de linux con ejemplos Kurt Wall 2000 Prentice Hall ISBN 987-9460-09-X