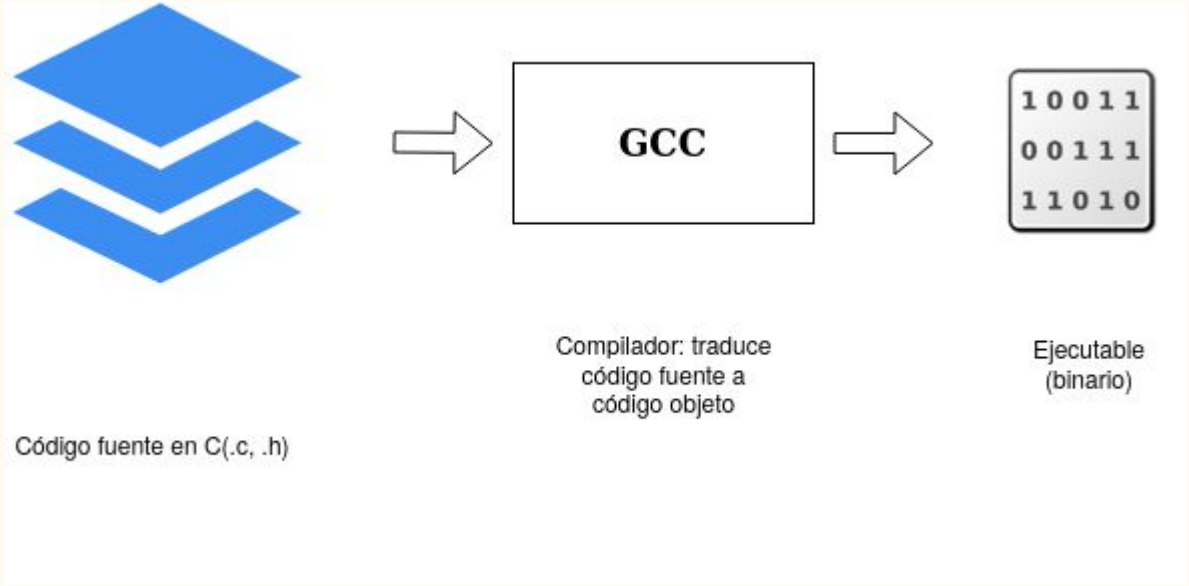
GDB: usando el debugger

Para entender qué está pasando con mi programa

GDB: Qué es esto? Con que se come?

- GDB es un debugger creado por Richard Stallman. Nos permite “inspeccionar” una ejecución de nuestro programa paso a paso consultando el valor de variables y expresiones a lo largo de una ejecución. También me permite obtener información detallada de la línea y archivo donde se produjo un fallo
- Se puede depurar código en C, C++, Java, etc.
- Además puede ser de gran utilidad en proyectos grandes para comprender lo que está ocurriendo en alguna sección crítica del código
- Es software libre, por ende tenemos acceso al código de fuente de gdb! (si nos interesa saber cómo funciona)

Cualquier programa se puede depurar con GDB?



Comenzando: agregar simbolos debugging

Lo primero que tenemos que hacer antes de comenzar a depurar un programa es compilarlo agregando los símbolos de debugging. A nuestras líneas habituales de compilación le agregamos el flag “-g”.

```
user@Laquesis:~/gdb_example$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -g -c array_helpers.c sort.c
```

```
user@Laquesis:~/gdb_example$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -g -o sorter *.o main.c
```

Tiene símbolos de debugging?

Para saber si un programa compilado o un archivo .o tiene símbolos de debugging podemos ejecutar: “**objdump** --syms <*ejecutable o .o*>| grep debug”. Si la salida del comando es vacía, no contiene símbolos de debugging, mientras que si muestra una tabla tendrá símbolos de debugging.

Consultando símbolos de debugging

- Sin símbolos:

```
user@Laquesis:/tmp/gdb_example$ gcc -Wall -Werror -Wextra -pedantic -std=c99  -c array_helpers.c sort.c
user@Laquesis:/tmp/gdb_example$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -o sorter *.o main.c
user@Laquesis:/tmp/gdb_example$ objdump --syms sorter | grep debug
user@Laquesis:/tmp/gdb_example$
```

- Con símbolos:

```
user@Laquesis:/tmp/gdb_example$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -g -c array_helpers.c sort.c
user@Laquesis:/tmp/gdb_example$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -g -o sorter *.o main.c
user@Laquesis:/tmp/gdb_example$ objdump --syms sorter | grep debug
0000000000000000 l    d  .debug_aranges 0000000000000000              .debug_aranges
0000000000000000 l    d  .debug_info    0000000000000000              .debug_info
0000000000000000 l    d  .debug_abbrev  0000000000000000              .debug_abbrev
0000000000000000 l    d  .debug_line    0000000000000000              .debug_line
0000000000000000 l    d  .debug_str     0000000000000000              .debug_str
```

Empezando a depurar mi programa

Para comenzar a depurar el programa que ya contiene símbolos de depuración tipeamos: “gdb ./<nombre_ejecutable>”

Respecto a los argumentos que toma el programa, tenemos 2 modos de ejecución:

1. Ejecutar el programa siempre con los mismos argumentos:

   “gdb --args ./<nombre_ejecutable> <args>”

2. Ejecutar el programa y decidir los argumentos dentro de gdb

   “gdb ./<nombre_ejectuable>” y dentro de gdb al comenzar la ejecución hacemos: “run <args>”

Argumentos fijos

```
user@Laquesis:/tmp/gdb_example$ gdb --args ./sorter input/empty.in
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./sorter...done.
(gdb) run
Starting program: /tmp/gdb_example/sorter input/empty.in
0
[Inferior 1 (process 13756) exited normally]
(gdb) quit
```

Argumentos variables

```
user@Laquesis:/tmp/gdb_example$ gdb ./sorter
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./sorter...done.
(gdb) run input/empty.in
Starting program: /tmp/gdb_example/sorter input/empty.in
0
[Inferior 1 (process 13779) exited normally]
(gdb) quit
```

Ejecutar un programa dentro de gdb

Para ejecutar el programa dentro de gdb una vez que hayamos iniciado como vimos anteriormente escribimos “**run**” y para salir de gdb podemos escribir “**quit**” o simplemente presionar “**CTRL + D**”

```
(gdb) run
Starting program: /tmp/gdb_example/sorter input/empty.in
0
[Inferior 1 (process 14150) exited normally]
(gdb) quit
```

Controlando el flujo de ejecución: breakpoints

Es posible colocar un “breakpoint” o punto de interrupción en cualquier lugar del código que necesitemos para que el programa detenga su ejecución al llegar a esta línea. Muy útil cuando quiero “investigar” qué está pasando al ejecutar determinada porción de código.

- Existen varias formas de poner un breakpoint
  - Usando el **nombre** de una función: **break** <nombre_de_funcion>**.** Si tenemos más de una función con el mismo nombre dentro del contexto del programa debemos aclarar el archivo también: **break** <archivo>:<nombre_funcion>
  - Usando la línea donde queremos detenernos: **break** <archivo>:<numero_linea>. Si ponemos número de línea solo sin especificar archivo gdb interpretará que es en el archivo donde se encuentra el main: **break** <numero_linea>

Controlando el flujo de ejecución: breakpoints

```
(gdb) b quick_sort
Breakpoint 1 at 0xf97: file sort.c, line 20.
(gdb) b sort.c:quick_sort_rec
Breakpoint 2 at 0xf38: file sort.c, line 12.
(gdb) b sort.c:14
Breakpoint 3 at 0xf57: file sort.c, line 14.
(gdb) b 15
Breakpoint 4 at 0xfca: file main.c, line 15.
```

Controlando el flujo de ejecución: breakpoints

En cualquier momento es posible:

- Consultar los breakpoints definidos: **info** breakpoints
- Habilitar un breakpoint: **enable** <breakpoint_number>
- Deshabilitar un breakpoint: **disable** <breakpoint_number>
- Borrar un breakpoint: **delete** <breakpoint_number>
- Los comandos habilitar, deshabilitar y borrar sin argumentos habilitarán, deshabilitarán o borrarán todos los breakpoints según corresponda. Ejemplo: "**delete**" borraría todos los breakpoints definidos

Controlando el flujo de ejecución: breakpoints

```
(gdb) break 8
Breakpoint 1 at 0xfca: file main.c, line 8.
(gdb) break sort.c:14
Breakpoint 2 at 0xf57: file sort.c, line 14.
(gdb) break quick_sort
Breakpoint 3 at 0xf97: file sort.c, line 20.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000fca in print_help at main.c:8
2       breakpoint     keep y   0x0000000000000f57 in quick_sort_rec at sort.c:14
3       breakpoint     keep y   0x0000000000000f97 in quick_sort at sort.c:20
(gdb) disable 3
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000fca in print_help at main.c:8
2       breakpoint     keep y   0x0000000000000f57 in quick_sort_rec at sort.c:14
3       breakpoint     keep n   0x0000000000000f97 in quick_sort at sort.c:20
(gdb) delete 1
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x0000000000000f57 in quick_sort_rec at sort.c:14
3       breakpoint     keep n   0x0000000000000f97 in quick_sort at sort.c:20
(gdb) delete
Delete all breakpoints? (y or n) y
```

Controlando el flujo de ejecución

Una vez que hemos establecido nuestros puntos de interrupción procedemos la ejecución con el comando **run.** Cada vez que el código pase por algún breakpoint la ejecución se detendrá y podremos controlar manualmente el flujo de ejecución con los comandos:

- **next**: Ejecuta la siguiente instrucción
- **step**: Igual que next pero permite “entrar” dentro de una función
- **continue**: Sigue el flujo de ejecución hasta el próximo breakpoint
- **finish**: ejecuta hasta finalizar la función actual

Controlando el flujo de ejecución

```
(gdb) break array_copy
Breakpoint 1 at 0x9a9: file array_helpers.c, line 7.
(gdb) run
Starting program: /tmp/gdb_example/sorter input/example-sorted.in

Breakpoint 1, array_copy (copy=0x7ffffff3a6a0, array=0x7ffffff9c130, length=5) at array_helpers.c:7
7           for (unsigned int i = 0u; i < length; ++i) {
(gdb) next
8               copy[i] = array[i];
(gdb) finish
Run till exit from #0  array_copy (copy=0x7ffffff3a6a0, array=0x7ffffff9c130, length=5) at array_helpers.c:8
main (argc=2, argv=0x7fffffffdd48) at main.c:61
61          quick_sort(array, length);
(gdb) step
quick_sort (a=0x7ffffff9c130, length=5) at sort.c:20
20          quick_sort_rec(a, 0u, (length == 0u) ? 0u : length - 1u);
(gdb) step
quick_sort_rec (a=0x7ffffff9c130, izq=0, der=4) at sort.c:12
12          if (izq < der) {
(gdb) continue
Continuing.
```

¿Donde estoy ?

Mientras ejecutó el programa en gdb cuando me detengo en un breakpoint a veces resulta útil saber en dónde estoy:

- **where** : dice en qué función, línea y archivo estoy.
- **list** <*n*>: imprime 10 líneas de código centradas en la línea *n*
- **list** -: imprime las 10 líneas anteriores
- **list** <nombre_función>: imprime 10 líneas centradas en la función seleccionada

¿Donde estoy ?

```
(gdb) run input/example-unsorted.in
Starting program: /tmp/gdb_example/sorter input/example-unsorted.in

Breakpoint 1, array_copy (copy=0x7ffffff3a690, array=0x7ffffff9c120, length=5) at array_helpers.c:7
7           for (unsigned int i = 0u; i < length; ++i) {
(gdb) where
#0  array_copy (copy=0x7ffffff3a690, array=0x7ffffff9c120, length=5) at array_helpers.c:7
#1  0x000055555555519b in main (argc=2, argv=0x7fffffffdd38) at main.c:59
(gdb) list 7
2       #include <stdbool.h>
3       #include <stdio.h>
4       #include <stdlib.h>
5
6       void array_copy(int copy[], int array[], unsigned int length) {
7           for (unsigned int i = 0u; i < length; ++i) {
8               copy[i] = array[i];
9           }
10      }
11
(gdb) list -
1       #include <assert.h>
(gdb) list array_copy
1       #include <assert.h>
2       #include <stdbool.h>
3       #include <stdio.h>
4       #include <stdlib.h>
5
6       void array_copy(int copy[], int array[], unsigned int length) {
7           for (unsigned int i = 0u; i < length; ++i) {
8               copy[i] = array[i];
9           }
10      }
```

Consultando el estado de las variables

Durante la detención en un breakpoint podemos consultar el estado de las variables en contexto:

- **ptype** <*var*>: imprime el tipo de la variable *var*
- **print** <*format*> <*var or expr*>: imprime el contenido de la variable *var* con el formato definido en *format*. Si no ponemos formato, usará uno predefinido

  Podemos también colocar un expresión que es evaluada antes de imprimir. Ejemplo: “**print** x+10”

Consultando el estado de las variables

```
(gdb) ptype length
type = unsigned int
(gdb) print length
$1 = 5
(gdb) print length+3
$2 = 8
(gdb) print length*4
$3 = 20
```

Cambiando el valor de las variables

Dentro de una ejecución podremos también cambiar el valor de una variable.
Advertencia: Este cambio altera la ejecución original del programa. Deberíamos realizar estos cambios sólo en situaciones “de laboratorio” para corroborar alguna hipótesis acerca de la falla.

```
(gdb) print length
$4 = 5
(gdb) set length = 0
(gdb) print length
$5 = 0
```

Moviéndonos por la pila de ejecución

En medio de una ejecución de un programa tenemos el contexto asociado de pila de ejecución. Cada vez que llamamos a una función en el código la función padre queda en stand by y se agrega a la pila la función llamada. Gdb permite consultar la pila de ejecución y desplazarnos por ella.

- **backtrace**: imprime el estado de la pila de ejecución
- **up**: sube un nivel en la pila de ejecución
- **down**: baja un nivel en la pila de ejecución
- **frame**: retorna en que nivel de la pila ejecución estoy
- **frame** $<n>$: avanza hasta el nivel n de la pila de ejecución

Moviéndonos por la pila de ejecución

```
(gdb) backtrace
#0  array_copy (copy=0x7ffffff3a690, array=0x7ffffff9c120, length=0) at array_helpers.c:7
#1  0x000055555555519b in main (argc=2, argv=0x7fffffffdd38) at main.c:59
(gdb) frame
#0  array_copy (copy=0x7ffffff3a690, array=0x7ffffff9c120, length=0) at array_helpers.c:7
7           for (unsigned int i = 0u; i < length; ++i) {
(gdb) backtrace
#0  array_copy (copy=0x7ffffff3a690, array=0x7ffffff9c120, length=0) at array_helpers.c:7
#1  0x000055555555519b in main (argc=2, argv=0x7fffffffdd38) at main.c:59
(gdb) up
#1  0x000055555555519b in main (argc=2, argv=0x7fffffffdd38) at main.c:59
59          array_copy(copy, array, length);
(gdb) down
#0  array_copy (copy=0x7ffffff3a690, array=0x7ffffff9c120, length=0) at array_helpers.c:7
7           for (unsigned int i = 0u; i < length; ++i) {
(gdb) fram 1
#1  0x000055555555519b in main (argc=2, argv=0x7fffffffdd38) at main.c:59
59          array_copy(copy, array, length);
(gdb) frame
#1  0x000055555555519b in main (argc=2, argv=0x7fffffffdd38) at main.c:59
59          array_copy(copy, array, length);
```

Controlando el flujo de ejecución: watchpoints

Los watchpoints son muy similares a los breakpoints con la diferencia que en lugar de frenar en un punto específico del código frenan cada vez que se modifica el valor de una variable.

- **watch** $<var>$: establece un punto de detención cada vez que la variable *var* sea modificada
- Podemos usar los mismos comandos que usábamos para los breakpoints: **enable**, **disable**, **delete**, **info watchpoints**

Controlando el flujo de ejecución: watchpoints

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /tmp/gdb_example/sorter input/example-unsorted.in

Breakpoint 1, array_copy (copy=0x7ffffff3a690, array=0x7ffffff9c120, length=5) at array_helpers.c:7
7           for (unsigned int i = 0u; i < length; ++i) {
(gdb) watchpoint i
Undefined command: "watchpoint".  Try "help".
(gdb) watch i
Hardware watchpoint 3: i
(gdb) n
8               copy[i] = array[i];
(gdb) c
Continuing.

Hardware watchpoint 3: i

Old value = 0
New value = 1
0x00005555555549de in array_copy (copy=0x7ffffff3a690, array=0x7ffffff9c120, length=5) at array_helpers.c:7
7           for (unsigned int i = 0u; i < length; ++i) {
(gdb) info watchpoints
Num     Type           Disp Enb Address            What
3       hw watchpoint  keep y                      i
        breakpoint already hit 1 time
```

Consultando la ayuda para un comando

Si quedaron dudas sobre qué puede hacer o no un comando de hdb siempre podremos consultar a su página de ayuda: **help** <*comando*>. Esto nos mostrará la ayuda detallada de dicho comando.

```
(gdb) help run
Start debugged program.
You may specify arguments to give it.
Args may include "*", or "[...]"; they are expanded using the
shell that will start the program (specified by the "$SHELL" environment
variable).  Input and output redirection with ">", "<", or ">>"
are also allowed.

With no arguments, uses arguments last specified (with "run" or
"set args").  To cancel previous arguments and run with no arguments,
use "set args" without arguments.

To start the inferior without using a shell, use "set startup-with-shell off".
```

Links útiles

- https://www.gnu.org/software/gdb/
- https://www.gnu.org/software/gdb/documentation/