Analisis avanzado de entornos de tiempo de ejecucion.

Elias Gill Quintana

"Estructura de los lenguajes"

Dr. Christian D. Von Lücken Martínez

Mayo, 2024

Introducción

El análisis y comprension del entorno de ejecución de los programas es un conocimiento clave para el desarrollo de software, ya que los distintos tipos de datos se representan de formas diferentes dentro de la memoria. Entender estas diferencias es importante a la hora de prevenir y/o resolver errores o cuellos de botella dentro del código.

Se discutirán conceptos esenciales como las pilas de llamadas, el tiempo de vida de las variables y el alcance de las mismas, y se demostrará a través de ejemplos prácticos, cómo GDB puede ser utilizado para inspeccionar y manipular estos elementos.

Este ensayo proporciona un análisis detallado de los entornos de ejecución de los programas y la representación de las distintas estructuras de datos en la memoria. Valiéndose del manejo del debugger GDB y ejemplos prácticos, se pretende comprender el ciclo de vida de la memoria de un programa.

Desarrollo

1. Preparando GDB

Para el desarrollo de los siguientes ejemplos utilizaremos el debugger GDB, el cual es un depurador para codigo C y C++. Compilaremos el codigo presentado en la especificacion de requerimientos del ensayo e iniciaremos GDB:

```
gcc -g programa.c -o main
gdb -q main
```

Con esto configuramos el compilador para generar informacion extra de depuracion dentro del archivo de salida de la compilacion. Esta informacion es utilizada por GDB para resolver y mostrar informacion legible para los humanos acerca del estado del programa.

Inicialmente colocaremos solo un punto de parada, esto se realiza utilizando el comando "break" dentro de GDB. Este punto sera: "recursive_function()". Con este punto de podremos analizar a detalle la pila de llamadas.

Cabe mencionar que un comando obligatorio para entender es el comando "help", el cual nos muestra los manuales detallados sobre las operaciones que el depurador es capaz de realizar.

2. La pila de llamadas

Iniciamos la ejecucion del programa con el comando "run", el programa se parara automaticamente en el primer punto de parada, el cual es la llamada a la funcion recursive_function().

Se nos mostrara una corta informacion del punto de parada. Se vera parecida a esto:

```
#0  recursive_function (n=0, item=0x5555555592f8) at code.c:28
28        static_recursive_var += 5;
```

Podremos ver a detalle la pila de llamadas[1] utilizando el comando "backtrace". Este comando nos listara el contenido de la pila, la cual esta compuesta por frames[2]. La informacion de cada frame la podemos ver de manera extensa y detallada utilizando el comando "info":

```
(gdb) info frame
   Stack level 0, frame at 0x7fffffffd020:
   rip = 0x55555555520c in recursive_function (code.c:26); saved rip = 0x555555555220
   called by frame at 0x7fffffffd050
   source language c.
   Arglist at 0x7fffffffd010, args: n=4, item=0x555555559330
   Locals at 0x7fffffffd010, Previous frame's sp is 0x7fffffffd020
   Saved registers:
   rbp at 0x7fffffffd010, rip at 0x7fffffffd018
```

[1] Tambien conocida como "call stack", es una estructura dinámica de datos la cual almacena la información sobre las subrutinas activas del programa.

[2] Los frames contienen información sobre la ejecución de la función, como las variables locales, los argumentos de la función, la dirección de retorno, entre otros detalles.

Tambie podemos movernos entre los distintos frames del stack utilizando el comando "frame", lo que nos permite visualizar informacion de los distintos frames. Esto sera de gran ayuda mas adelante cuando analicemos lo que ocurre con los distintos tipos de variables.

3. Llamadas recursivas

Si continuamos la ejecucion del programa utilizando el comando "continue", este se parara cada vez que se alcancemos nuestro break point. Si repetimos este procedimiento un par de veces y luego ejecutamos el comando "backtrace", podremos ver el contenido de la pila de ejecucion:

```
#0  recursive_function (n=0, item=0x5555555592f8) at code.c:18
#1  0x0000555555555220 in recursive_function (n=1, item=0x5555555592f8) at code.c:26
#2  0x0000555555555220 in recursive_function (n=2, item=0x5555555592f8) at code.c:26
#3  0x0000555555555220 in recursive_function (n=3, item=0x5555555592f8) at code.c:26
#4  0x0000555555555220 in recursive_function (n=4, item=0x5555555592f8) at code.c:26
#5  0x0000555555555220 in recursive_function (n=5, item=0x5555555592f8) at code.c:26
#6  0x000055555555539f in main () at code.c:58
```

Se puede observar que la pila ahora contiene las entradas de las distintas llamadas que fueron ocurriendo de manera recursiva. Si ahora ejecutamos el comando "info frame" sobre dos frames distintos, podremos observar mas a detalle como los espacios de memoria son distintos dentro de las funciones:

```
Frame 1:
called by frame at 0x7fffffffd090, caller of frame at 0x7fffffffd030
Arglist at 0x7fffffffd050, args: n=5, item=0x5555555592c0
Locals at 0x7fffffffd050, Previous frame's sp is 0x7fffffffd060
Saved registers:
rbp at 0x7fffffffd050, rip at 0x7fffffffd058

Frame 2:
called by frame at 0x7fffffffd060, caller of frame at 0x7fffffffd000
Arglist at 0x7fffffffd020, args: n=4, item=0x5555555592c0
Locals at 0x7fffffffd020, Previous frame's sp is 0x7fffffffd030
Saved registers:
rbp at 0x7fffffffd020, rip at 0x7fffffffd028
```

Aqui podemos observar como las distintas llamadas recursivas cuentan con su propia de memoria, en el cual se encuentran sus propias variables locales y argumentos.

3.1. Variables locales Aprovechando el estado actual de nuestro debugger, podemos averiguar que pasa con las variables locales de las funciones.

Primeramente podemos movernos al frame perteneciente a la funcion "main" y tratar de acceder a las variables locales de nuestra "recursive_function":

```
(gdb) info address local_var
No symbol "local_var" in current context.
```

Como es de esperar, intentar esto resulta en un error. Esto debido a que la variable pertenece al frame de ejecucion de "recursive_function", no al de "main". A este concepto lo conocemos como "scope". Si se desea analizar mas a detalle las variables que forman parte del scope de un frame especifico, en ese caso podemos hacer uso de la funcion "info scope".

Del mismo modo podemos observar como las mimas variables pero de frames distintos tienen a su vez valores distintos. Ya sabemos que esto se debe a que los frames tienen espacios de memoria aislados entre si.

Todas la variables locales pertenecientes a un frame especifico son automaticamente "destruidas" (desalojadas) al termino de la ejecucion de dicho frame, liberando asi el espacion de memoria correspondiente a dichas variables. Por tanto, el intento de acceder o utilizar dichos espacios luego de su destruccion puede traer consecuencias nefastas para nuestro programa.

3.2. Variables estaticas

Un caso curioso ocurre con la variables estaticas. Estas son un tipo especial de variables que se tiene un comportamiento "parecido" a las variables globales, en el sentido de que su posicion de memoria ya queda definida en tiempo de compilacion, pero a diferencia de las variables globales, estas variables estaticas si cuentan con un alcance limitado al scope en donde se encuentren definidas, es decir, solo pueden ser accedidas desde las funciones donde fueron definidas (o pasadas como parametro).

Esto implica que cualquier modificacion realizada en un frame que tenga dentro de su scope a la variable estatica tambien afectara a los demas frames que tambien tengan acceso a dicha variable.

Esto queda mas claro con un ejemplo. Para ello utilizaremos una caracteristica de GDB la cual es la capacidad de modificar variables en tiempo de ejecucion de manera arbitraria. Por ejemplo, tomemos la variable estatica en un frame modifiquemos su valor. Esto se realiza con el comando "set":

```
Frame 1:
(gdb) set static_recursive_var = 192
(gdb) info locals
static_recursive_var = 192

Frame 2:
(gdb) info locals
local_var = 192
```

Claramente podemos notar de que la modificacion de la variable estatica en el frame 1 afecta directamente al valor en el frame 2. Esto ocurre dado que la direccion de memoria de la variable estatica ya estaba definida en tiempo de compilacion, por tanto el estado de esta variable es compartido entre los frames que tengan acceso a su valor.

Ademas, aun podemos conocer la informacion acerca de su **posicion en la memoria** desde cualquier frame (dado a su caracter estatico), por mas de que en realidad esta variable es innacesible desde el scope de dicho frame.

```
(gdb) info variables static_recursive_var
   All variables matching regular expression "static_recursive_var":
   Non-debugging symbols:
   0x0000555555558040  static_recursive_var

(gdb) info address static_recursive_var
No symbol "static_recursive_var" in current context.

(gdb) info scope main
   Scope for main:
   Symbol main_var is a complex DWARF expression:
   0: DW_OP_fbreg -40
   , length 4.
   Symbol dynamic_var is a complex DWARF expression:
   0: DW_OP_fbreg -32
   , length 8.
   Symbol items is a complex DWARF expression:
   0: DW_OP_fbreg -24
   , length 8.
```

3.3. Variables globales

Las variables globales en cambio son variables las cuales su posicion de memoria ya esta definida en tiempo de compilacion, y a su vez pueden ser accedidas desde cualquier punto del programa en ejecucion. Su modificacion tiene la mismas consecuencias que en las variables estaticas. Para esto GDB cuenta con el comando "info variables":

```
(gdb) info variables global_var
All variables matching regular expression "global_var":
  File code.c:
  4:    int global_var;

(gdb) info address global_var
Symbol "global_var" is static storage at address 0x555555558038.
```

Tanto las variables globales como las variables estaticas al estar definidas en tiempo de compilacion no se liberan, por tanto estas tienen un tiempo de vida igual a la duracion de la ejecucion del programa, a diferencia de sus hermanas las variables locales, las cuales tienen un tiempo de vida igual al tiempo de ejecucion de la funcion donde fueron creadas.

ESTRUCTURAS DE DATOS DINAMICAS Y PASO POR PARAMETRO

Tabla de contenidos