# Análisis Avanzado de Entornos de Tiempo de Ejecución con GDB

**Alexis Rodrigo Arce Delgadillo**

**Estructura de los Lenguajes**

**Profesor: Christian Daniel von Lücken Martínez**

Junio 2024

# Índice

## Resumen

Este ensayo analiza en profundidad los entornos de tiempo de ejecución utilizando el depurador GDB (GNU Debugger). A través de un programa en C con variables locales, globales, estáticas y dinámicas, así como funciones recursivas, se exploran la pila de llamadas, el paso de parámetros y las estructuras de datos dinámicas. Se destaca cómo GDB facilita la comprensión del comportamiento interno de los programas, permitiendo identificar y corregir errores de manera efectiva. Este análisis muestra la utilidad de GDB para mejorar la eficiencia y robustez de los programas.


## 1. Introducción

El análisis profundo de los entornos de tiempo de ejecución es fundamental para comprender el funcionamiento interno de los programas informáticos. En este contexto, el uso del depurador GDB (GNU Debugger) se ha vuelto indispensable para los desarrolladores en la identificación y corrección de errores, así como para la optimización de programas en lenguajes como C. Este ensayo se centra en el análisis avanzado de entornos de tiempo de ejecución mediante el uso de GDB como herramienta principal.

El objetivo principal de este ensayo es proporcionar una visión detallada de cómo GDB puede ser utilizado para explorar y comprender los distintos aspectos de un programa en ejecución. Desde la visualización de la pila de llamadas hasta la manipulación de variables locales y globales, pasando por el análisis de estructuras de datos dinámicas, este ensayo abordará las diversas funcionalidades que GDB ofrece para la depuración y análisis de programas.

Para ilustrar estos conceptos, se empleará un programa en C que incluye elementos como variables locales, globales, estáticas y dinámicas, funciones recursivas y estructuras de datos dinámicas. A través de la configuración de puntos de interrupción estratégicos y el uso de comandos específicos de GDB, se explorará el comportamiento del programa en diferentes contextos de ejecución, permitiendo una comprensión más profunda de los entornos de tiempo de ejecución.

Este ensayo no solo se enfocará en la utilización práctica de GDB, sino también en la importancia de comprender los fundamentos teóricos detrás de los entornos de tiempo de ejecución, como la diferencia en el alcance y tiempo de vida de las variables, el paso de parámetros a funciones y la gestión de memoria dinámica. A través de este análisis, se busca destacar la relevancia de GDB como una herramienta esencial en el proceso de desarrollo de software, promoviendo la eficiencia, la calidad y la fiabilidad de los programas desarrollados.

## 2. Desarrollo

Para realizar el análisis avanzado de entornos de tiempo de ejecución utilizando GDB, seguimos varios pasos importantes. A continuación, se describe el proceso de preparación del entorno, compilación del programa, y configuración de puntos de interrupción en GDB.

### 2.1. Preparación del Entorno

Primero, preparamos el entorno de desarrollo instalando GDB en una máquina Linux o en el WSL (Subsistema de Windows para Linux). Esto se logra ejecutando el siguiente comando en la terminal:

```
sudo apt-get install gdb
```

### 2.2. Compilación del programa

Luego, procedemos a compilar el programa en C con opciones de depuración para permitir una inspección detallada durante la ejecución. Utilizamos el siguiente comando de compilación:

```
gcc -g -o debug_example debug_example.c
```

Entonces el comando `gcc -g -o debug_example debug_example.c` se desglosa de la siguiente forma:

- Usar el compilador `gcc`
- Incluir la información de depuración en el ejecutable (opción `-g`)
- Generar un ejecutable llamado `debug_example` (opción `-o debug_example`)
- Compilar el archivo fuente `debug_example.c`

### 2.3. Inicio de GDB y configuración de puntos de interrupción

Una vez compilado el programa, iniciamos GDB con el ejecutable recién creado:

```
gdb ./debug_example
```

Dentro de GDB, configuramos puntos de interrupción para instruir al depurador que pause la ejecución del programa en ubicaciones específicas. Estas ubicaciones pueden ser líneas de código, funciones particulares o condiciones específicas. En nuestro caso, vamos a establecer puntos de interrupción en las siguientes funciones críticas que queremos analizar: `modify_static_var`, `recursive_function`, `process_items` y main. Esto se hace utilizando los siguientes comandos:

```
(gdb) break modify_static_var
(gdb) break recursive_function
(gdb) break process_items
(gdb) break main
```

Con los puntos de interrupción configurados, iniciamos la ejecución del programa dentro de GDB utilizando el comando run:

```
(gdb) run
```

### 2.4. Exploración de Frames y Pila de Llamadas con GDB

El programa se ejecuta y GDB detendrá la ejecución cuando alcance el primer punto de interrupción, en nuestro caso la función main. Para continuar la ejecución del programa hasta el siguiente punto de interrupción, utilizamos el comando continue o simplemente c:

```
(gdb) continue
```

Cuando la ejecución se detiene en un punto de interrupción, podemos explorar los frames y la pila de llamadas para entender el contexto de la ejecución en ese momento. Utilizamos el comando **backtrace** o **bt** para mostrar la pila de llamadas, que muestra todas las funciones activas desde la más reciente hasta la inicial por ejemplo:

```
(gdb) backtrace
#0  process_items (items=0x7ffffffee030, count=0) at debug_example.c:26
#1  0x00000000080013bf in main () at debug_example.c:45
(gdb)
```

En este caso, se usó de ejemplo el punto de interrupción `process_items` para explorar un frame específico, usamos el comando **frame N**, donde **N** es el número del frame que queremos inspeccionar:

```
(gdb) frame 0
#0  process_items (items=0x7ffffffee030, count=0) at debug_example.c:26
26      void process_items(Item *items, int count) {
(gdb) frame 1
#1  0x00000000080013bf in main () at debug_example.c:45
45          process_items(items, 3);
```

Esto nos permite ver y examinar las variables locales y los parámetros de función en ese frame particular, lo que proporciona una visión detallada del estado del programa en diferentes puntos. La numeración en la pila de llamadas de gdb indica el orden de las funciones desde la más reciente (actualmente en ejecución) hasta la más antigua (la base de la pila).

Para obtener información específica sobre el frame seleccionado, podemos utilizar el comando **info frame**. Por ejemplo con el punto de interrupción actual, si ejecutamos **info frame** después de seleccionar un frame específico con **frame N**, obtendríamos información similar a la siguiente:

```
(gdb) frame 1
#1  0x00000000080013bf in main () at debug_example.c:45
45          process_items(items, 3);
(gdb) info frame
```

```
Stack level 1, frame at 0x7fffffee040:
 rip = 0x80013bf in main (debug_example.c:45); saved rip = 0x7fffff5d4083
 caller of frame at 0x7fffffee010
 source language c.
 Arglist at 0x7fffffee008, args:
 Locals at 0x7fffffee008, Previous frame's sp is 0x7fffffee040
 Saved registers:
 rbp at 0x7fffffee030, rip at 0x7fffffee038
```

Esto nos proporciona información detallada sobre los argumentos de la función, las variables locales y los registros de la CPU en ese contexto específico de ejecución.

### 2.5. Diferencia en el Tiempo de Vida y Alcance de las Variables

- **Variables Locales:** Tienen un alcance limitado al bloque o función en el que se declaran. Su tiempo de vida comienza cuando se entra en el bloque o función y termina cuando se sale de él.

  Para examinar el comportamiento de las variables locales podemos utilizar el punto de interrupción **recursive_function**. Luego, utilizamos el comando **continue** para reanudar la ejecución del programa y permitir que se realicen varias llamadas recursivas.

  ```
  (gdb) c
  Continuing.
  Recursion depth: 3, Local Variable: 3, Static Recursive Variable: 50,
  Global Variable: 100, Item ID: 1, Item Name: Item_1

  Breakpoint 2, recursive_function (n=0, item=0x0) at debug_example.c:16
  16      void recursive_function(int n, Item *item) {
  ```

  En este punto, la ejecución se detiene en la función **recursive_function**. Podemos usar el comando info locals para observar las variables locales:

  ```
  (gdb) info locals
  local_var = 0
  static_recursive_var = 50
  ```

  Continuamos con el siguiente comando:

  ```
  (gdb) n
  17          int local_var = n;
  ```

  Avanzamos a la siguiente línea de código con el comando **next** (n):

  ```
  (gdb) n
  19          if (n > 0) {
  ```

Volvemos a verificar las variables locales:

```
(gdb) info locals
local_var = 2
static_recursive_var = 50
```

Podemos ver que **local_var** ha cambiado a 2, lo que refleja el nuevo valor asignado en el contexto de la ejecución actual. A continuación, se detalla cómo cambiar el valor de una variable en tiempo de ejecución usando GDB. En este caso, intentamos modificar el valor de la variable n dentro de la función recursiva:

```
(gdb) set var n = -5
```

Después de modificar el valor de n, continuamos la ejecución y observamos cómo los valores de las variables cambian en consecuencia:

```
(gdb) n
17          int local_var = n;
(gdb) info locals
local_var = -5
static_recursive_var = 50
```

Aquí, el valor de **local_var** refleja el nuevo valor de **n** que hemos asignado, demostrando cómo se puede modificar el estado del programa en tiempo de ejecución utilizando GDB.

- **Variables Estáticas:** Mantienen su valor entre llamadas a la función y tienen un alcance local a la función en la que se definen.

  Para observar el comportamiento de una variable estática, establecemos un punto de interrupción en la función **modify_static_var** y comenzamos la ejecución del programa. Aquí está el proceso detallado utilizando GDB:

```
(gdb) c
Continuing.
Starting the program...
Block Variable: 30

Breakpoint 1, modify_static_var () at debug_example.c:11
11      void modify_static_var() {
(gdb) info locals
static_var = 10
```

  Continuamos la ejecución para ver cómo cambia el valor de **static_var** después de la primera llamada a la función:

```
(gdb) c
Continuing.
Static Variable: 20

Breakpoint 1, modify_static_var () at debug_example.c:11
11      void modify_static_var() {
(gdb) info locals
static_var = 20
```

El valor de **static_var** ha aumentado a 20, lo que demuestra que mantiene su estado entre llamadas a la función. Continuamos una vez más para ver el valor después de otra llamada:

```
(gdb) c
Continuing.
Static Variable: 30
```

El valor de **static_var** ha aumentado a 30, demostrando nuevamente que su estado se conserva entre llamadas sucesivas a la función **modify_static_var**. Este comportamiento de las variables estáticas es útil en situaciones donde se necesita mantener información entre llamadas a una función sin utilizar variables globales. Permite que los datos sean persistentes a lo largo de la vida del programa mientras se limitan a un ámbito local, protegiéndolos de accesos no autorizados o modificaciones accidentales desde otras partes del código.

- **Variables Globales:** Tienen alcance en todo el programa y existen durante toda la ejecución. Su tiempo de vida es desde que se inicia el programa hasta que termina. Para ilustrar el manejo de las variables globales, utilizamos el depurador GDB para observar y modificar la variable global **global_var** en el programa. A continuación, se muestra el proceso:

  ```
  (gdb) print global_var
  $1 = 100
  ```

  Continuando con la ejecución por ejemplo en el punto de interrupción **modify_static_var**:

  ```
  (gdb) c
  Continuing.
  Recursion depth: 5, Local Variable: 5, Static Recursive Variable: 50,
  Global Variable: 100, Item ID: 1, Item Name: Item_1

  Breakpoint 2, recursive_function (n=32767, item=0x0) at debug_example.c:16
  16      void recursive_function(int n, Item *item) {
  ```

  El valor de **global_var** sigue siendo 100. Ahora, vamos a modificar su valor en tiempo de ejecución para ver cómo afecta el comportamiento del programa:

```
(gdb) set var global_var = 200
(gdb) print global_var
$4 = 200
```

El valor de **global_var** ha cambiado a 200. Continuamos la ejecución para ver el impacto de este cambio:

```
(gdb) c
Continuing.
Recursion depth: 3, Local Variable: 3, Static Recursive Variable: 50,
Global Variable: 200, Item ID: 1, Item Name: Item_1

Breakpoint 2, recursive_function (n=0, item=0x0) at debug_example.c:16
16      void recursive_function(int n, Item *item) {
```

Podemos observar que el valor de **global_var** reflejado en la salida de la función recursiva es ahora 200, lo que confirma que el cambio ha sido aplicado exitosamente en tiempo de ejecución.

- **Variables Dinámicas:** Su alcance depende de cómo se maneje la memoria y su tiempo de vida termina cuando se libera la memoria asignada.

  Para demostrar la manipulación de las variables dinámicas, hemos empleado el depurador GDB para examinar y modificar las variables **dynamic_var** y la estructura **Item** en el programa. A continuación, se presenta el procedimiento:

  Una vez detenido en el punto de interrupción **main** inspeccionamos el valor de la variable dinámica **dynamic_var** y de la estructura **Item** usando el comando **print**:

```
(gdb) print *dynamic_var
$2 = 20
(gdb) print items[0]
$3 = {id = 0, name = '\000' <repeats 49 times>}
```

Estos valores aún no han sido inicializados correctamente. Continuamos la ejecución hasta llegar a la función **process_items** donde se inicializan las estructuras dinámicas:

```
(gdb) c
Continuing.
Static Variable: 30

Breakpoint 3, process_items (items=0x7ffffffee030, count=0) at debug_example.c
26      void process_items(Item *items, int count) {
```

En este punto, podemos observar cómo se inicializan los elementos de **items**:

```
(gdb) print items[0]
$5 = {id = 1, name = "Item_1", '\000' <repeats 43 times>}
(gdb) print items[1]
$6 = {id = 2, name = "Item_2", '\000' <repeats 43 times>}
(gdb) print items[2]
$7 = {id = 3, name = "Item_3", '\000' <repeats 43 times>}
```

Continuamos la ejecución y verificamos el uso de **dynamic_var** dentro de la función recursiva **recursive_function**:

```
(gdb) c
Continuing.

Breakpoint 2, recursive_function (n=0, item=0x308001450) at debug_example.c:16
16      void recursive_function(int n, Item *item) {

(gdb) n
Recursion depth: 5, Local Variable: 5, Static Recursive Variable: 50,
Global Variable: 100, Item ID: 1, Item Name: Item_1
21              recursive_function(n - 1, item);
(gdb) print *item
$8 = {id = 1, name = "Item_1", '\000' <repeats 43 times>}
```

El primer elemento de items tiene id 1 y nombre **Item_1**.

Finalmente, continuamos la ejecución hasta el final del programa, donde se liberan las variables dinámicas:

```
(gdb) print dynamic_var
$1 = (int *) 0x80052a0
(gdb) print items
$2 = (Item *) 0x80052c0
```

El uso de variables dinámicas permite una gestión más flexible de la memoria, especialmente útil cuando no se conoce de antemano el tamaño de los datos que se manejarán. Sin embargo, es crucial gestionar adecuadamente la asignación y liberación de memoria para evitar errores como fugas de memoria o accesos a memoria no válida.

## 3. Conclusiones

En conclusión, el análisis avanzado de entornos de tiempo de ejecución utilizando el depurador GDB (GNU Debugger) ofrece una visión profunda y detallada del comportamiento interno de los programas en ejecución. A lo largo de este ensayo, hemos explorado cómo GDB facilita la comprensión y depuración de programas en C, centrándonos en varios aspectos clave:

La capacidad de GDB para explorar frames y la pila de llamadas proporciona una visión cronológica y estructurada de cómo se ejecutan las funciones en un programa, permitiendo un análisis exhaustivo del flujo de ejecución y la secuencia de eventos.

Hemos observado la diferencia en el tiempo de vida y alcance de las variables locales, estáticas, globales y dinámicas, destacando la importancia de comprender cómo se manejan estas variables en diferentes contextos de ejecución.

El paso de parámetros a funciones y la posibilidad de modificarlos en tiempo de ejecución con GDB nos permite realizar pruebas y experimentos controlados para comprender mejor el comportamiento de nuestras funciones y programas.

También hemos explorado cómo GDB facilita el manejo de estructuras de datos dinámicas, como la asignación y liberación de memoria, ayudando a prevenir fugas de memoria y optimizar el uso de recursos.

En conjunto, el uso efectivo de GDB en el análisis de entornos de tiempo de ejecución no solo mejora la capacidad de depuración de los desarrolladores, sino que también promueve una comprensión más profunda y completa de cómo funcionan y se comportan los programas durante su ejecución. Esta comprensión es esencial para desarrollar software robusto, eficiente y confiable, y GDB se posiciona como una herramienta indispensable en el arsenal de cualquier desarrollador que busque mejorar la calidad y el rendimiento de sus programas en C.