# Depuración de programas utilizando GDB

La depuración necesita que nuestro código fuente sea compilado con –g para crear una tabla de símbolos mejorada. Por tanto, el siguiente comando:

$ gcc -g file1.c file2.c -o prog

hará que se cree prog con símbolos de depuración en su tabla de símbolos. Si queremos, podemos utilizar la opción –ggdb de GCC para generar aún más información de depuración (específica de GDB). Sin embargo, para funcionar de manera más efectiva, esta opción requiere que tengamos acceso al código fuente de todas las bibliotecas con las que enlazamos. Mientras que esto puede ser muy útil en ciertas situaciones, también puede ser caro en términos de espacio de disco. Sin embargo, en la mayoría de los casos, deberíamos poder arreglárnoslas con la opción –g sencilla.

En el capítulo 8 del libro [Wall], se señala que es posible utilizar ambas opciones, -g y –O (optimización). Sin embargo, como la optimización cambia el programa resultante, puede que la relación que esperamos entre el código que escribimos y el binario ejecutable no exista. Es posible que las variables o líneas de código hayan desaparecido o puede que las asignaciones de las variables se produzcan cuando no lo esperamos. Por tanto, debemos esperar hasta haber depurado nuestro código tan completamente como sea posible, antes de empezar a optimizarlo. A la larga hará que nuestra vida, especialmente la parte de ella que pasamos depurando, sea mucho más sencilla y menos estresante.

## Cómo utilizar los comandos GDB básicos

La mayor parte de lo que necesitamos conseguir con GDB puede hacerse con un conjunto de comandos sorprendentemente pequeño. A continuación, se muestran se muestran suficientes comandos GDB para empezar.

### Cómo iniciar GDB

Para iniciar una sesión de depuración, hay que escribir sencillamente gdb progname [archivo de núcleo], (la utilización del archivo de núcleo es opcional) remplazando progname con el nombre del programa que queremos depurar. Para ejemplificar una sesión de depuración se utilizará el siguiente listado

Listado 8.1 ([Wall])

/\*

\*Debugme.c - Programa pobremente escrito para depurar.

\*/

#include <stdio.h>

#include <stdlib.h>

#define BIGNUM 5000

void index\_to\_the\_moon(int ary[ ]);

int main(void)

{

int intary[100];

index\_to\_the\_moon(intary);

exit(EXIT\_SUCCESS);

}

void index\_to\_the\_moon(int ary[ ])

{

int i;

for (i = 0; i < BIGNUM; ++i)

ary[i] = i;

}

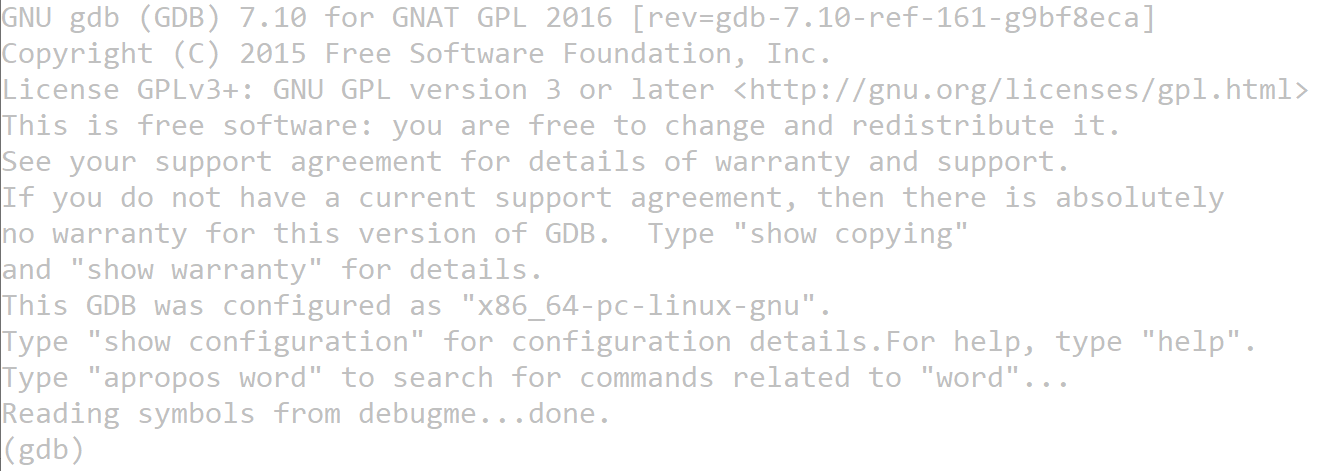
Compilamos este programa utilizando make debugme.

En la mayoría de los sistemas, si intentamos ejecutarlo escribiendo ./debugme, provocará inmediatamente un error de segmentación y un volcado de núcleo.

El primer paso es iniciar GDB utilizando el nombre del programa, debugme, y el nombre de archivo de núcleo, core, como argumentos:

$ gdb debugme core

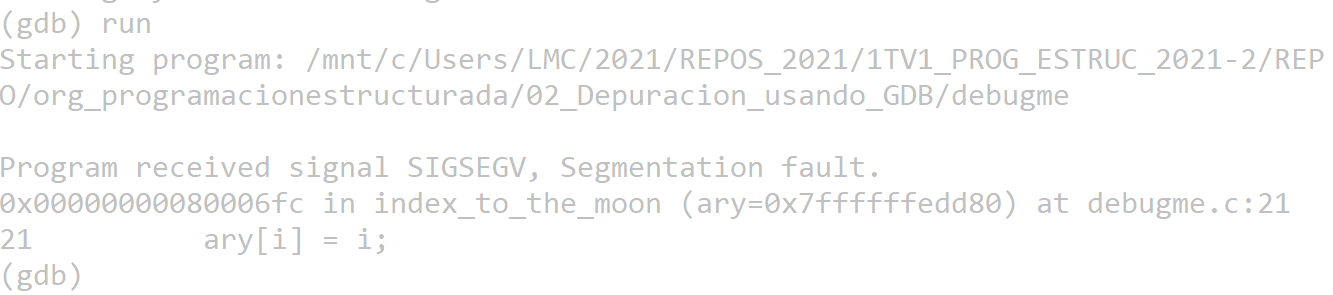
La pantalla debería parecerse a la siguiente figura



Pantalla de inicio de GDB

Si no nos gustan los mensajes de licencia, podemos utilizar la opción –q (o --quiet) para suprimirlos. Otra opción de línea de comandos útil es –d dirname, donde dirname es el nombre de un directorio. Esto le dice a GDB dónde encontrar el código fuente (por omisión, mira en el directorio que esté actualmente en funcionamiento).

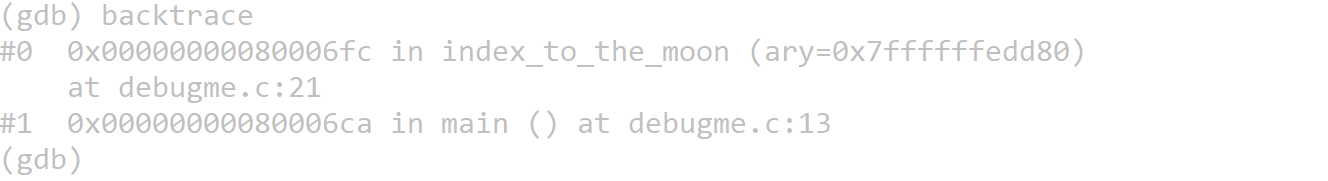
Lo primero que deberíamos hacer es ejecutar el programa en el depurador. El comando para hacer esto es run. Al intentar ejecutar el programa en el depurador, GDB para después recibir la señal SIGSEGV:



Este corto listado de salida muestra que el error de segmentación se produjo en la función index\_to\_the\_moon en la línea 21 de debugme.c. Observemos que GDB muestra la línea de código culpable. También nos ofrece la dirección en la que se produjo el fallo.

## Inspección de código en el depurador

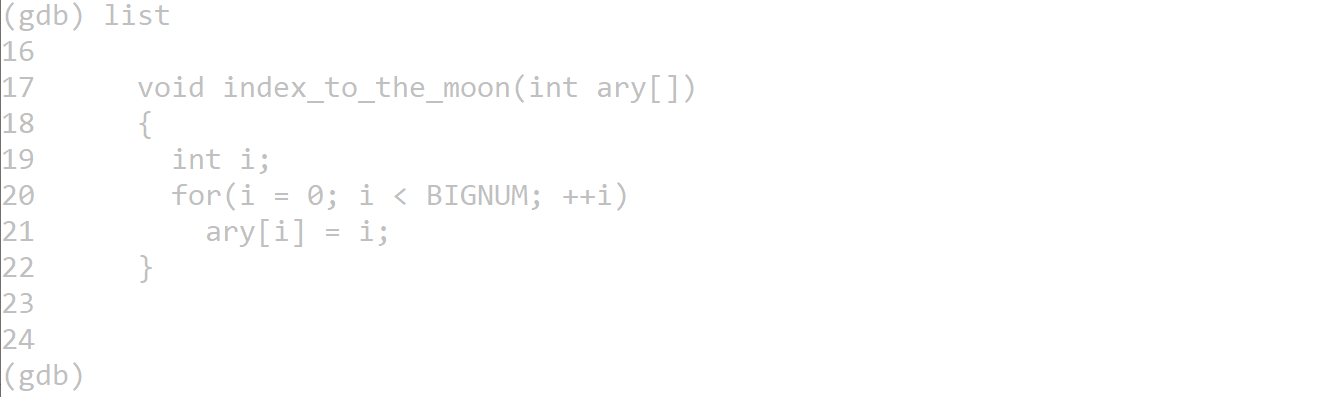
La pregunta es, ¿qué estaba ocurriendo en la función index\_to\_the\_moon? Podemos ejecutar el comando backtrace para generar el árbol de funciones que llevó al error de segmentación. El rastro de la función (para este programa en nuestro sistema) es como sigue:



En GDB no es necesario los nombres de los comandos completos. Cualquier abreviatura que sea única servirá. Por ejemplo, back será suficiente para backtrace.

Por tanto, el problema estaba en index\_to\_the\_moon, que como muestra gdb, fue llamada desde la línea 13 de la función main de debugme.c.

Sin embargo, sería de ayuda tener alguna idea del contexto en el que está la línea(s) de código que ha causado controversia. Para este propósito, utilizamos el comando list, que toma la forma general list [m,n]. m y n son los números de línea inicial y final que queremos que se visualicen. Un comando list sencillo mostrará el código que rodea el área donde se detectó inicialmente el error, como podemos ver en lo siguiente:



Con una idea clara de lo que está ocurriendo en el código y dónde, podemos decidir qué es lo que ha salido mal y como arreglarlo.

## Cómo examinar los datos

Una de las características de GDB más útiles es su capacidad de visualizar el tipo y el valor de casi cualquier expresión, variable o matriz de un programa que esté siendo depurado. Imprimirá el valor de cualquier expresión legal en el lenguaje en el que está escrito nuestro programa. El comando es, por supuesto, print. A continuación tenemos algunos comandos print y sus resultados:

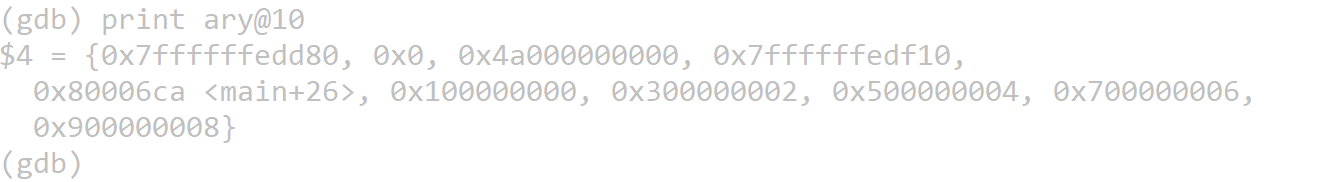


Este ejemplo continúa los ejemplos anteriores de depuración debugme.c: aún estamos intentando identificar por qué y dónde falló debugme. Aunque en este ejemplo, el programa se estropeó cuando la variable contador i igualó 1184, donde falle en nuestro sistema dependerá de la distribución de memoria del sistema, el espacio de memoria de proceso, la cantidad de memoria disponible en el sistema y otros factores.

El segundo comando, print ary[i], deja bastante claro que el programa no tiene acceso a la localidad de memoria especificada, aunque sí tiene acceso legal a la anterior. El $1 y $2 se refieren a entradas en la historia de valores. Si queremos acceder a estos valores en el futuro, utilizaremos estos alias en lugar de volver a escribir el comando. Por ejemplo, el comando $1 – 1 produce:



No estamos limitados a la utilización de valores discretos, porque gdb puede mostrar las direcciones de los datos almacenados en una región arbitraria de memoria. Para imprimir las diez primeras localidades de memoria asociadas con ary, utilizamos el siguiente comando:



La notación @10 significa imprimir los 10 valores que empiezan en ary. Por el contrario, digamos que queremos imprimir los cinco valores almacenados en ary comenzando con el primer elemento. El comando sería el siguiente:



Como cada comando print crea una nueva entrada en la historia de comandos de GDB, podemos utilizar los valores de matriz agregados más adelante.

Quizá nos estamos preguntando por qué el penúltimo comando print mostraba valores hexadecimales y el último mostraba valores decimales. En primer lugar, recordemos que las matrices de C están basadas en cero. Recordemos también que el nombre de la matriz es un apuntador a la base de la misma. Por tanto, GDB miró en ary, vio que era la dirección de la base de la matriz y la mostró junto con los nueve valores siguientes como direcciones de memoria. Las direcciones de memoria normalmente se visualizan en hexadecimal. Si queremos visualizar los diez primeros valores almacenados en ary, utilizaremos el operador índice [ ], con el índice del primer valor, 0, como se ilustra a continuación:



GDB también puede decirnos los tipos de variables con el comando whatis:



Esta característica puede parecer bastante inútil porque, por supuesto, ya conocemos los tipos de todas las variables de nuestro programa (¡si claro!). Pero cambiaremos de opinión la primera vez que tengamos que depurar el código de otra persona o tengamos que arreglar un proyecto multiarchivo y no hayamos mirado uno de los archivos fuente durante un par de meses.

## Cómo configurar puntos de ruptura

Cuando depuramos código problemático, con frecuencia resulta de utilidad detener la ejecución en algún punto. GDB nos permite establecer puntos de ruptura en varias clases diferentes de construcciones de código, incluyendo números de línea y nombres de función, y también nos permite configurar puntos de ruptura condicionales donde el código solo se para si se cumple cierta condición.

Para configurar un punto de ruptura en un número de línea, utilizamos la siguiente sintaxis:

(gdb) break linenum

Para detener la ejecución cuando el código se introduce en una función, utilizamos

(gdb) break funcname

En cualquier caso, GDB detendrá la ejecución antes de ejecutar el número de línea especificado o de introducir la función especificada. Después, podemos, por ejemplo, utilizar print para visualizar los valores de variable o utilizar list para revisar el código que se va a ejecutar. Si tenemos un proyecto multiarchivo y queremos detener la ejecución en una línea de código o en una función que no está en el archivo fuente actual, utilizamos lo siguiente:

(gdb) break filename:linenum

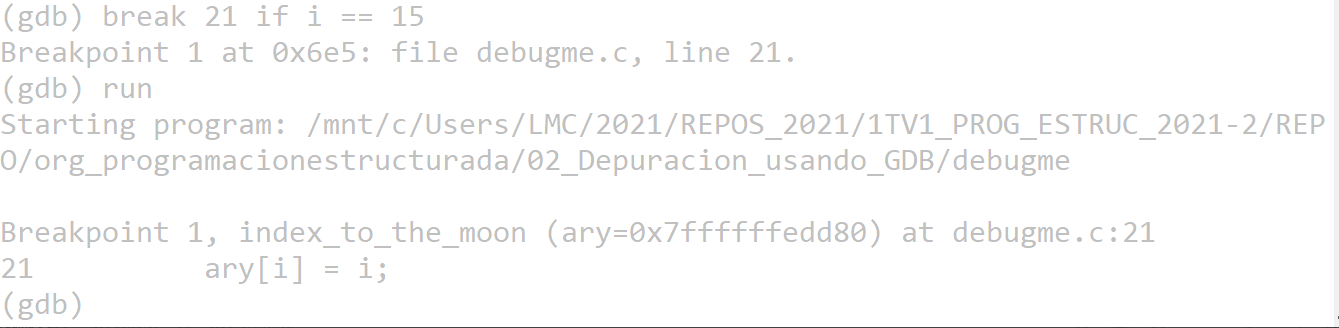
(gdb) break filename:funcname

Los puntos de ruptura condicionales son, normalmente, más útiles. Se llaman así porque nos permiten detener la ejecución de un programa temporalmente si o cuando se cumpla una condición en particular. La sintaxis correcta para configurar los puntos de ruptura condicionales es:

(gdb) break linenum if expr

(gdb) break funcname if expr

Expr puede ser cualquier expresión que evalúe como verdad (no cero). Por ejemplo, el siguiente comando break detiene la ejecución en la línea 24 de debugme cuando la variable i es igual a 15:



Como podemos ver, GDB se paró en la línea 21. Un rápido comando print confirma que se detuvo cuando el valor de i alcanzó el valor solicitado:



Si el programa ya se está ejecutando cuando introducimos el comando run, GDB nos dirá que el programa ya se ha iniciado y nos pregunta si queremos reiniciarlo desde el principio. Digamos sí.

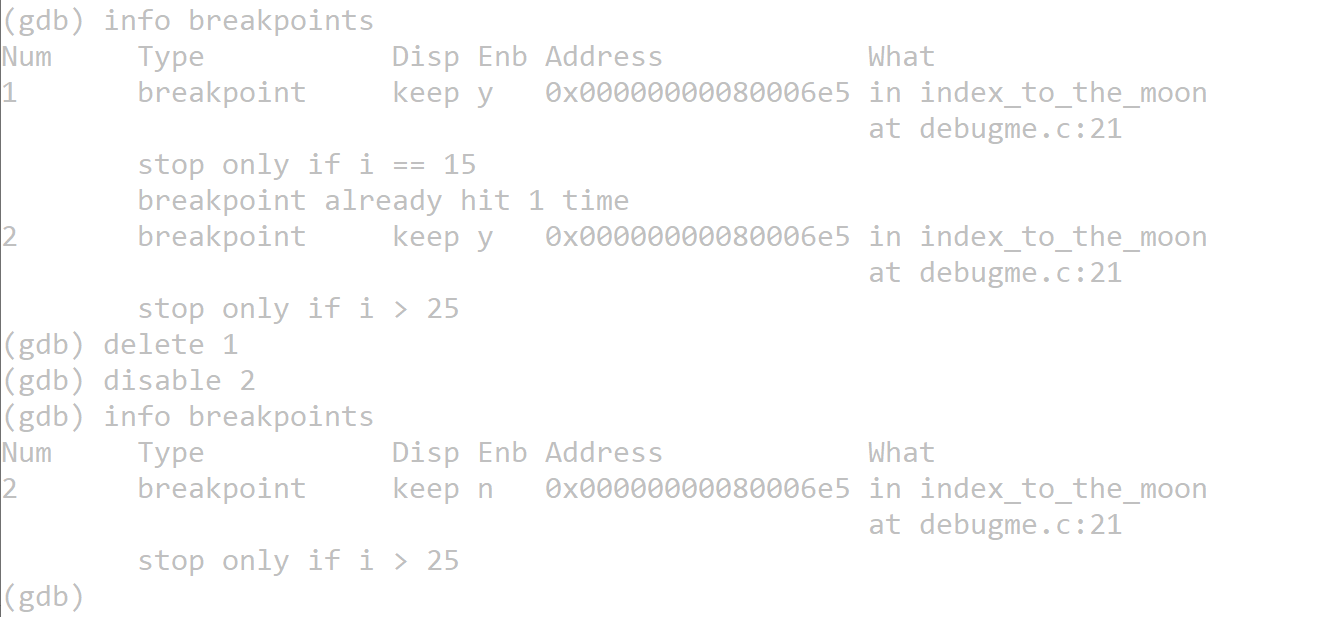
Para reanudar la ejecución después de un punto de ruptura, escribamos sencillamente continue. Si hemos establecido muchos puntos de ruptura y hemos perdido la pista de lo que se ha configurado y de cuáles se han disparado, podemos utilizar el comando info breakpoints para refrescarnos la memoria. El comando delete nos permite eliminar puntos de ruptura, o podemos simplemente desactivarlos. La figura de abajo ilustra la salida de los siguientes comandos

(gdb) info breakoints

(gdb) delete 1

(gdb) disable 2

(gdb) info breakpoints



Administración de los puntos de ruptura GDB

En la figura de arriba, el comando info breakpoints enumera los puntos de ruptura disponibles. Después de eliminar el primer punto de ruptura y desactivar el segundo, otro comando info breakpoints volvió a mostrar la información de los puntos de ruptura. No nos sorpenderá que el comando para reactivar un punto de ruptura sea enable N, donde N es el número de punto de ruptura.

REFERENCIA

[Wall] Kurt Wall, el al. Programación en Linux al Descubierto, 2/a Edición, Prentice Hall, 2001.