Arquitectura de sistemas

Abelardo Pardo

University of Sydney
School of Electrical and Information Engineering
NSW, 2006, Australia
Autor principal del curso de 2009 a 2012

Iria Estévez Ayres

Damaris Fuentes Lorenzo

Pablo Basanta Val

Pedro J. Muñoz Merino

Hugo A. Parada

Derick Leony

Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática
Avenida Universidad 30, E28911 Leganés (Madrid), España

Capítulo 14. El programa depurador o *debugger*

Tabla de contenidos

Uno de los principales problemas al desarrollar aplicaciones son los errores de ejecución. Compilar un programa no es garantía suficiente de que funciona de la manera prevista. Es más, el ciclo de desarrollo de un programa está ocupado, en su mayoría por las tareas de diagnosticar y corregir los errores de ejecución. A los errores de ejecución en programas se les suele denominar en inglés *bugs* (bichos).

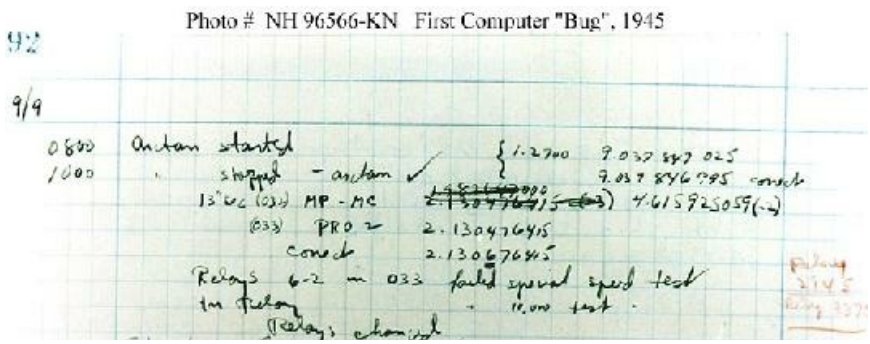
El origen de la utilización del término *bug* para describir los errores en un programa es un poco confuso, pero hay dos referencias documentadas a las cuales se les suele atribuir este mérito.

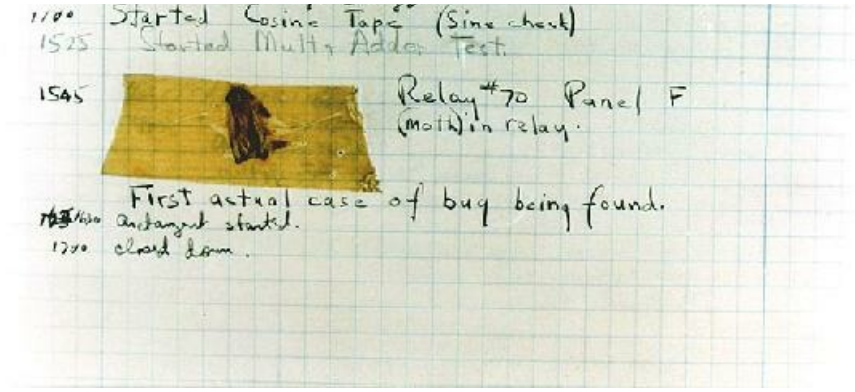
La primera de ellas se puede encontrar en una carta que Edison escribió en 1878 en la que se puede leer:

> It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise -- this thing gives out and [it is] then that "Bugs" -- as such little faults and difficulties are called -- show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached.

Fuente: Edison to Puskas, 13 de noviembre de 1878, Edison papers, Edison National Laboratory, U.S. National Park Service, West Orange, N.J., citado en "Thomas P. Hughes, American Genesis: A History of the American Genius for Invention", Penguin Books, 1989, página 75.

La invención del término se atribuye generalmente a la ingeniera Grace Hopper que en 1946 estaba en el laboratorio de computación de la universidad de Harvard trabajando en los ordenadores con nombre Mark II y Mark III. Los operadores descubrieron que la causa de un error detectado en el Mark II era una polilla que se había quedado atrapada entre los contactos de un relé (por aquel entonces el elemento básico de un ordenador) que a su vez era parte de la lógica interna del ordenador. Estos operadores estaban familiarizados con el término *bug* e incluso pegaron el insecto en su libro de notas con la anotación "First actual case of bug being found" (primer caso en el que realmente se encuentra un *bug*) tal y como ilustra la siguiente figura (Fuente: U.S. Naval Historical Center Photograph)





Hoy en día, los métodos que se utilizan para "depurar" los errores de un programa son múltiples y con diferentes niveles de eficacia. El método consistente en insertar llamadas a `printf` que escriben en pantalla mensajes es quizás el más ineficiente de todos ellos. En realidad lo que se precisa es una herramienta que permita ejecutar de forma controlada un programa, que permita suspender la ejecución en cualquier punto para poder realizar comprobaciones, ver el contenido de las variables, etc.

Esta herramienta se conoce con el nombre de depurador o, su término inglés, *debugger*. El depurador es un ejecutable cuya misión es permitir la ejecución controlada de un segundo ejecutable. Se comporta como un envoltorio dentro del cual se desarrolla una ejecución normal de un programa, pero a la vez permite realizar una serie de operaciones específicas para visualizar el entorno de ejecución en cualquier instante.

Más concretamente, el depurador permite:

- ejecutar un programa línea a línea
- detener la ejecución temporalmente en una línea de código concreta
- detener temporalmente la ejecución bajo determinadas condiciones
- visualizar el contenido de las variables en un determinado momento de la ejecución
- cambiar el valor del entorno de ejecución para poder ver el efecto de una corrección en el programa

Uno de los depuradores más utilizados en entornos Linux es ***gdb*** (Debugger de GNU). En este documento se describen los comandos más relevantes de este depurador para ser utilizados con un programa escrito en C. Todos los ejemplos utilizados en el resto del documento se basan en el programa cuyo código fuente se muestra a continuación (y que está en el fichero gdb_use.c).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define SIZE 1000
5
6  struct data_unit
7  {
8    int data;
9    struct data_unit *next;
10 };
11 typedef struct data_unit unit, *unit_ptr;
12
13 void function(unit_ptr table)
14 {
15   int y;
```

```
16    for (y = 0; y < SIZE - 1; y++)
17    {
18      table[y].data = y;
19      table[y].next = &table[y + 1];
20    }
21    table[SIZE - 1].data = SIZE - 1;
22  }
23
24  int check(unit_ptr table)
25  {
26    int y;
27    for (y = 0; y < SIZE; y++)
28    {
29      if ((table[y].data + 1) != table[y].next->data)
30      {
31        return 1;
32      }
33    }
34    return 0;
35  }
36
37  int main(int argc, char **argv)
38  {
39    unit_ptr buf;
40
41    buf = (unit_ptr)calloc(SIZE, sizeof(unit_ptr));
42    function(buf);
43    if (check(buf))
44    {
45      printf("Error detectado en tabla\n");
46    }
47    free(buf);
48
49    return 0;
50  }
```

14.1. Arranque y parada del depurador

Para que un programa escrito en C pueda ser manipulado por **gdb** es preciso realizar una compilación que incluya como parte del ejecutable, un conjunto de datos adicionales. Esto se consigue incluyendo la opción **-g** al invocar el compilador:

```
$ gcc -Wall -g -o gdb_use gdb_use.c
```

Una vez creado el fichero ejecutable se invoca el depurador con el comando:

```
$ gdb gdb_use
```

Tras arrancar el depurador se muestra por pantalla un mensaje seguido del prompt **(gdb)**:

```
$ gdb gdb_use
GNU gdb (GDB) 7.0-ubuntu
Copyright (C) 2009 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.
```

```
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 "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/test/gdb_use...done.
(gdb)
```

En este instante, el programa depurador ha arrancado, pero la ejecución del programa **gdb_use** (que se ha pasado como primer argumento) todavía no. La interacción con el depurador se realiza a través de comandos introducidos a continuación del prompt, de forma similar a como se proporcionan comandos a un *shell* o intérprete de comandos en Linux.

Para arrancar la ejecución del programa se utiliza el comando **run** (o su abreviatura **r**). Tras introducir este comando, el programa se ejecuta de forma normal. Si se produce un error o una interrupción en la ejecución, el programa se detiene y el control vuelve al depurador, por lo que se muestra por pantalla de nuevo el prompt **(gdb)**. Por ejemplo:

```
(gdb) r
Starting program: /home/test/gdb_use

Program received signal SIGSEGV, Segmentation fault.
0x08048441 in check (table=0x804a008) at gdb_use.c:29
29          if ((table[y].data + 1) != table[y].next->data)
(gdb)
```

Cuando se produce un error, el depurador muestra el lugar en el código responsable de ese error. Es posible que el error se produzca en una rutina interna del sistema. En tal caso, no quiere decir que el error esté en tal librería, sino que se ha manifestado en la librería por causas debidas al código del programa.

Si se desea detener un programa mientras se está ejecutando se debe pulsar Crtl-C (la tecla control, y mientras se mantiene pulsada, se pulsa C). La interrupción del programa es capturada por el depurador, y el control lo retoma su intérprete de comandos. En este instante, la ejecución del programa ha sido detenida pero no terminada. Prueba de ello, es que la ejecución puede continuarse mediante el comando **continue** (que se puede abreviar simplemente con la letra **c**).

Para salir del depurador se utiliza el comando **quit** (abreviado por la letra **q**). Si se pretende terminar la sesión del depurador mientras el programa está en ejecución se pide confirmación para terminar dicha ejecución.

```
(gdb) q
The program is running.  Exit anyway? (y or n) y
$
```

El comando **help** muestra la información referente a todos los comandos y sus opciones. Si se invoca sin parámetros, se muestran las categorías en las que se clasifican los comandos. Si el comando va seguido del nombre de una categoría, proporciona información detallada sobre los comandos que contiene. Si se invoca seguido de un comando, describe su utilización.

14.2. La pila de llamadas

El comando **where** muestra en cualquier instante de la ejecución lo que se denomina la pila de llamadas. Esta pila no es más que la secuencia de funciones que están siendo ejecutadas en ese instante. Por ejemplo:

```
(gdb) r
Starting program: /home/test/gdb_use

Program received signal SIGSEGV, Segmentation fault.
0x08048441 in check (table=0x804a008) at gdb_use.c:29
29          if ((table[y].data + 1) != table[y].next->data)
(gdb) where
#0  0x08048441 in check (table=0x804a008) at gdb_use.c:29
#1  0x0804849f in main (argc=1, argv=0xfefff604) at gdb_use.c:43
(gdb)
```

El mensaje anterior muestra cómo el programa se ha detenido en la función `check`, concretamente en la línea 29, que a su vez ha sido invocada por la función `main` en la línea 43. Nótese que estas líneas aparecen en orden inverso al que se producen. La primera llamada en producirse, obviamente, ha sido la llamada en la función `main`, línea 43.

Por defecto, el depurador se detiene en el código de la última llamada. Pero a menudo es preciso moverse a través de esa pila y situarse en una llamada anterior. Los comandos **up** y **down** permiten moverse por la pila de llamada y seleccionar cualquiera de ellas. La utilidad de estos comandos será más aparente cuando se utilice en combinación con los comandos de visualización de datos.

```
(gdb) where
#0  0x08048441 in check (table=0x804a008) at gdb_use.c:29
#1  0x0804849f in main (argc=1, argv=0xfefff604) at gdb_use.c:43
(gdb) up
#1  0x0804849f in main (argc=1, argv=0xfefff604) at gdb_use.c:43
43        if (check(buf))
(gdb) down
#0  0x08048441 in check (table=0x804a008) at gdb_use.c:29
29          if ((table[y].data + 1) != table[y].next->data)
(gdb)
```

14.3. Visualización de código

El código fuente del programa en ejecución se puede mostrar por pantalla mediante el comando **list** (abreviado **l**). Sin opciones, este comando muestra la porción de código alrededor de la línea que está siendo ejecutada en el instante en el que se detuvo. Si el programa no está en ejecución, se muestra la rutina `main`. Este comando acepta opciones para mostrar una línea en concreto, una línea en un fichero, una función en un fichero, e incluso el código almacenado en una dirección de memoria completa. El comando **help list** muestra todas las opciones posibles.

```
(gdb) l main
32          }
33        }
34        return 0;
33      }
36
37      int main(int argc, char **argv)
38      {
39        unit_ptr buf;
40
41        buf = (unit_ptr)calloc(SIZE, sizeof(struct data_unit));
42        function(buf)
(gdb)
```

14.4. Ejecución controlada de un programa

Aparte de detener la ejecución de un programa con Crtl-C, lo más útil es detener la ejecución en una línea concreta del código. Para ello es preciso insertar un punto de parada (en inglés *breakpoint*). Dicho punto es una marca que almacena el depurador, y cada vez que la ejecución del programa pasa por dicho punto, suspende la ejecución y devuelve el control al usuario. Para insertar un punto de parada se utiliza el comando **break** (abreviado **b**) seguido de la línea en la que se desea introducir.

```
(gdb) l 41
36
37      int main(int argc, char **argv)
38      {
39        unit_ptr buf;
40
41        buf = (unit_ptr)calloc(SIZE, sizeof(struct data_unit));
42        function(buf);
43        if (check(buf))
44        {
(gdb) b 41
Breakpoint 1 at 0x8048471: file gdb_use.c, line 41.
(gdb)
```

Se pueden introducir tantos puntos de parada como sean necesarios en diferentes lugares del código. El depurador asigna un número entero a cada uno de ellos comenzando por el 1. En la última línea del mensaje anterior se puede ver como al punto introducido en la línea 41 del fichero **gdb_use.c** se le ha asignado el número 1.

El comando **info breakpoints** (o su abreviatura **info b**) muestra por pantalla la lista de puntos de parada que contiene el depurador.

```
(gdb) l 42
37      int main(int argc, char **argv)
38      {
39        unit_ptr buf;
40
41        buf = (unit_ptr)calloc(SIZE, sizeof(struct data_unit));
42        function(buf);
43        if (check(buf))
44        {
45          printf("Error detectado en tabla\n");
(gdb) b 42
Breakpoint 2 at 0x8048486: file gdb_use.c, line 42.
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048471 in main at gdb_use.c:41
2   breakpoint     keep y   0x08048486 in main at gdb_use.c:42
(gdb)
```

Los puntos de parada se pueden introducir en cualquier momento de la ejecución de un proceso. Una vez introducidos, si se comienza la ejecución del programa mediante el comando **run** (o su abreviatura **r**), ésta se detiene en cuanto se ejecuta una línea con un punto de parada.

```
(gdb) r
Starting program: /home/test/gdb_use
```

```
Starting program: /home/test/gdb_use

Breakpoint 1, main (argc=1, argv=0xfefff604) at gdb_use.c:41
41        buf = (unit_ptr)calloc(SIZE, sizeof(struct data_unit));
(gdb)
```

Los puntos de parada se pueden poner utilizando el nombre de una función. En tal caso la ejecución se detiene justo antes de ejecutar la primera línea de código de dicha función.

```
(gdb) b check
Breakpoint 3 at 0x80483ce: file gdb_use.c, line 27.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/test/gdb_use

Breakpoint 1, main (argc=1, argv=0xfefff604) at gdb_use.c:41
41        buf = (unit_ptr)calloc(SIZE, sizeof(struct data_unit));
(gdb) c
Continuing.

Breakpoint 2, main (argc=1, argv=0xfefff604) at gdb_use.c:42
42        function(buf);
(gdb) c
Continuing.

Breakpoint 3, check (table=0x804a008) at gdb_use.c:27
27        for (y = 0; y < SIZE; y++) {
(gdb)
```

Tal y como muestra el listado anterior, el depurador primero se ha detenido en el punto de parada 1. Tras introducir el comando **continue** se ha detenido en el punto de parada 2. Al ejecutar el comando **continue** de nuevo, la ejecución continua hasta la primera línea de la función `check`. Además de mostrar la línea de código en la que se ha detenido, también se muestra el valor de los parámetros con los que ha sido invocada la función. A menudo esta información es crucial para identificar la procedencia de un error. El comando **where** también ofrece esta información, pero para todas las funciones en ejecución en ese instante.

Cada punto de parada puede ser temporalmente desactivado/activado de manera independiente. Los comandos **enable** y **disable** seguido de un número de punto de parada activan y desactivan respectivamente dichos puntos.

Para reanudar la ejecución del programa previamente suspendida hay tres comandos posibles. El primero que ya se ha visto es **continue** (o **c**). Este comando continua la ejecución del programa y no se detendrá hasta que se encuentre otro punto de parada, se termine la ejecución, o se produzca un error. El segundo comando para continuar la ejecución es **next** (o su abreviatura **n**). Este comando ejecuta únicamente la línea de código en el que está detenido el programa y vuelve de nuevo a suspender la ejecución.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/test/gdb_use
```

```
Breakpoint 1, main (argc=1, argv=0xfefff604) at gdb_use.c:41
41        buf = (unit_ptr)calloc(SIZE, sizeof(struct data_unit));
(gdb) n

Breakpoint 2, main (argc=1, argv=0xfefff604) at gdb_use.c:42
42        function(buf);
(gdb) n
43        if (check(buf)) {
(gdb)
```

Nótese que si la línea de código procesada con el comando **next** es una invocación a una función, el depurador ejecuta la función y se detiene en la línea siguiente. Por tanto, este comando no permite introducirse en una llamada a una función. Para ese cometido existe el comando **step** (o su abreviatura **s**). Este comando, cuando se ejecuta sobre una línea de código que contiene una llamada a una función, en lugar de pasar a la siguiente línea, se introduce en la función y suspende de nuevo la ejecución.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/test/gdb_use

Breakpoint 1, main (argc=1, argv=0xfefff604) at gdb_use.c:41
41        buf = (unit_ptr)calloc(SIZE, sizeof(struct data_unit));
(gdb) n

Breakpoint 2, main (argc=1, argv=0xfefff604) at gdb_use.c:42
42        function(buf);
(gdb) s
function (table=0x804a008) at gdb_use.c:16
16        for (y = 0; y < SIZE - 1; y++)
(gdb)
```

Si el depurador está detenido sobre una línea que no contiene ninguna llamada a una función, el efecto de los comandos **next** y **step** es idéntico. Con el comando **where** se puede comprobar como el programa se ha detenido en la invocación de la función `function` desde la función `main`.

```
(gdb) where
#0  function (table=0x804a008) at gdb_use.c:16
#1  0x08048491 in main (argc=1, argv=0xfefff604) at gdb_use.c:42
(gdb)
```

14.5. Visualización de datos

Los comandos descritos hasta ahora permiten una ejecución controlada de un programa, pero cuando el depurador es realmente eficiente es cuando hay que localizar un error de ejecución. Generalmente, ese error se manifiesta como una terminación abrupta (por ejemplo `segmentation fault`). Cuando el programa se ejecuta desde el depurador, esa terminación retorna el control al depurador con lo que es posible utilizar comandos para inspeccionar el estado en el que ha quedado el programa.

Uno de los comandos más útiles del depurador es **print** (o su abreviatura **p**). Como argumento recibe una ***expresión***, y su efecto es imprimir el valor resultante de evaluar dicha expresión. Este comando puede recibir el nombre de cualquier variable que esté visible en ese instante en la ejecución del

programa. El contenido de una de estas variables se muestra por pantalla simplemente escribiendo el comando seguido del nombre.

```
Breakpoint 2, main (argc=1, argv=0xfefff604) at gdb_use.c:42
42        function(buf);
(gdb) s
function (table=0x804a008) at gdb_use.c:16
16        for (y = 0; y < SIZE - 1; y++)
(gdb) n
18          table[y].data = y;
(gdb) n
19          table[y].next = &table[y + 1];
(gdb) n
16        for (y = 0; y < SIZE - 1; y++)
(gdb) p table[0].data
$1 = 0
(gdb) p table[0].next
$2 = (struct data_unit *) 0x804a010
(gdb)
```

Aparte de nombres de variables, **print** acepta expresiones aritméticas en las que pueden aparecer nombres de variables (por ejemplo: **print x + 2**) o direcciones de memoria.

```
(gdb) p *table[0].next
$3 = {data = 0, next = 0x0}
(gdb) p table[0].next->data
$4 = 0
(gdb)
```

Nótese que el último comando print tiene como parámetro la expresión **table[0].next->data**. El depurador evalua esta expresión de forma ***idéntica*** a como lo hace el compilador, con los valores existentes en ese instante e imprime el resultado.

Algunas expresiones pueden ser útiles para conocer en detalle qué está sucediendo en un programa. Por ejemplo, si en un programa en C se reserva espacio para una tabla de 100 enteros, la línea correcta a escribir es:

```
tabla = (int *)malloc(100 * sizeof(int));
```

El depurador puede evaluar la expresión **sizeof(int)** y mostrar su resultado. Esta es una forma de saber el tamaño con el que trabaja el compilador en la plataforma que se está utilizando. En el programa de ejemplo se puede ver el tamaño que ocupa la estructura de datos **data_unit** mediante el comando:

```
(gdb) p sizeof(struct data_unit)
$5 = 8
(gdb)
```

14.6. Puntos de parada condicionales

A pesar de ofrecer la posibilidad de ejecución paso a paso y visualización de cualquier variable en el programa, para localizar errores de ejecución de forma efectiva, a menudo es preciso detenerse en un lugar del programa, pero bajo ciertas condiciones. Por ejemplo, supongamos que el código contiene un bucle con un número elevado de iteraciones. Los comandos **next** y **step** permiten realizar una ejecución paso a paso de cada iteración. Pero si el error de ejecución requiere un número elevado de

ejecución paso a paso de cada iteración. Pero si el error de ejecución requiere un número elevado de iteraciones antes de manifestarse, esta ejecución paso a paso es muy larga y tediosa. En esta situación, los comandos **next** y **step** no son tan útiles.

Un primer paso para el diagnóstico del problema consiste en ejecutar el programa sin ningún tipo de punto de parada. En cuanto la anomalía se manifiesta, el depurador suspende la ejecución en ese punto, y a través del comando **print** se pueden visualizar los valores de todas las variables visibles en ese instante. Combinando este comando con los comando **up** y **down** se pueden visualizar todas las variables activas en las diferentes funciones.

En un número elevado de casos, se detecta alguna variable que contiene un valor incorrecto y es la causa directa de la anomalía. Sin embargo, en general lo que se necesita saber no es qué variable ha causado la anomalía, sino cómo tal variable ha llegado a tener tal valor. En realidad, lo que se necesita es ejecutar el programa y detener su ejecución ***antes*** de que ejecute ciertas líneas de código en las que se produce el valor que acaba produciendo el error.

Para ilustrar esta situación se procede a desactivar todos los puntos de parada a la vez en el programa ejemplo mediante el comando **disable** sin argumentos. El comando **info breakpoints** muestra el valor **n** en el campo **Enb** (abreviatura de *Enable*) de todos los puntos.

```
(gdb) disable
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep n   0x08048471 in main at gdb_use.c:41
        breakpoint already hit 1 time
2   breakpoint     keep n   0x08048486 in main at gdb_use.c:42
        breakpoint already hit 1 time
3   breakpoint     keep n   0x0804843d in check at gdb_use.c:27
(gdb)
```

A continuación se ejecuta el programa y se puede observar cómo se produce un error en la línea 29 de código, más concretamente en la función `check`.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/test/gdb_use

Program received signal SIGSEGV, Segmentation fault.
0x08048441 in check (table=0x804a008) at gdb_use.c:29
29          if ((table[y].data + 1) != table[y].next->data)
(gdb)
```

La línea del error está incluida dentro de un bucle, cuyo índice es la variable `y`. Para ver en qué iteración se ha producido el error, se imprime el valor de esta variable.

```
(gdb) p y
$6 = 999
(gdb)
```

Se puede comprobar que el error se ha producido en la última de las iteraciones. Evidentemente, mediante los comandos **next** y **step** se puede llegar a detener la ejecución antes de que se produzca, pero tal tarea acarrearía introducir demasiados comandos todos idénticos.

Para facilitar esta labor **gdb** permite la definición de puntos de parada condicionales. Estos puntos se

introducen añadiendo a un punto de parada definido anteriormente una condición booleana que si evalua a cierto hace que la ejecución se detenga, mientras que si evalua a falso, el punto de parada no tiene efecto alguno.

Para conseguir que el programa se detenga antes de ejecutar la línea 29 es necesario introducir dos comandos.

```
(gdb) b 29
Breakpoint 4 at 0x804844f: file gdb_use.c, line 29.
(gdb) condition 4 y == 999
(gdb)
```

El primer comando crea un punto de parada incondicional en la línea 29 del fichero `gdb_use.c`. Esta línea corresponde a la comprobación de la condición en el bucle de la función `check`. El comando **condition** añade al punto de parada número 4 la condición **y == 999**. Nótese que la sintaxis de la condición es directamente código C. A partir de ese momento se puede comprobar la definición del punto de parada mediante el comando **info breakpoints**.

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep n   0x080484a1 in main at gdb_use.c:41
2   breakpoint     keep n   0x080484b6 in main at gdb_use.c:42
3   breakpoint     keep n   0x0804843d in check at gdb_use.c:27
4   breakpoint     keep y   0x0804844f in check at gdb_use.c:29
 stop only if y == 999
(gdb)
```

De los cuatro puntos de parada definidos, los tres primeros están desactivados tal y como indica el cuarto campo, y el último está activado pero se detiene sólo si **y == 999**.

La inserción de un punto de parada condicional se puede realizar mediante un único comando si se añade al comando **breakpoint** la palabra **if** seguida por una condición. En el ejemplo anterior, el punto de parada se puede insertar mediante el comando **b 29 if (y == 999)**.

Si se ejecuta de nuevo el programa, esta vez la ejecución se detiene justo antes de producirse el error.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/test/gdb_use

Breakpoint 4, check (table=0x804a008) at gdb_use.c:29
29     if ((table[y].data + 1) != table[y].next->data)
(gdb)
```

Si el error se produce en esta línea, se procede a visualizar mediante el comando **print** el contenido de las variables y expresiones que forman parte de ella.

```
(gdb) p table[y].data + 1
$2 = 1000
(gdb) p table[y].next
$3 = (struct data_unit *) 0x0
(gdb) p table[y].next->data
```

```
Cannot access memory at address 0x0
(gdb)
```

Se puede observar que se realiza una indirección a través del campo **next** del elemento de la tabla cuyo valor es 0 y por tanto una dirección incorrecta. En este punto se ha localizado el error. Falta descubrir si ese campo tiene el valor correcto, y si es así, por qué se utiliza como indirección (ejercicio que se deja para que lo resuelva el lector).

Si se desea borrar una condición de un punto de parada simplemente se añade una condición vacía (comando **condition** seguido únicamente del número de punto de parada).

```
(gdb) condition 4
Breakpoint 4 now unconditional.
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep n   0x080484a1 in main at gdb_use.c:41
2   breakpoint     keep n   0x080484b6 in main at gdb_use.c:42
3   breakpoint     keep n   0x0804843d in check at gdb_use.c:27
4   breakpoint     keep y   0x0804844f in check at gdb_use.c:29
 breakpoint already hit 1 time
(gdb)
```

14.7. Preguntas de autoevaluación

Comprueba con estas preguntas que has entendido cómo funciona el depurador.

1. Tras crear un fichero ejecutable, `./program`, e invocar al depurador con el comando `gdb program`, quieres ejecutar el programa mediante el depurador. ¿Cuál es el comando correcto?

   - `run` ó `r`
   - `start` ó `s`
   - `continue` ó `c`

2. Una vez arrancada la ejecución del programa con el depurador, quieres introducir un punto de ruptura en la línea 36 de tu código. ¿Cuál es el comando correcto?

   - `l 36`
   - `b l 36`
   - `b 36`
   - `p 36`
   - Primero tienes que parar la ejecución para poder introducir cualquier punto de ruptura.

3. Teniendo en cuenta el siguiente fragmento de código del programa `calculate_square.c`:

```
5  int calculate_square(int n)
6  {
7    /* Function to print the square of a number */
8    int square = 0;
9    if (n<=250)
```

```
10   {
11     square = n * n;
12     printf(''The square of %d is %d \n", n, square);
13   }
14   return square;
15 }
16 int main(int argc, char **argv)
17 {
18   calculate_square(5);
19   calculate_square(251);
20   return 0;
21 }
```

Invocas el depurador, introduces un punto de ruptura en la llamada a la función de la línea 19 y arrancas el programa. El programa se detiene en la línea 19, y quieres entrar dentro de la función `calculate_square` para ver su funcionamiento cuando el argumento es 251, así es que:

- Ejecutas `c` para que el depurador continúe y entre dentro de la función.
- Ejecutas `n` para que el depurador continúe y entre dentro de la función.
- Ejecutas `s` para que el depurador continúe y entre dentro de la función.
- Ejecutas `c` para que el depurador continúe y ejecutas `n` para que entre dentro de la función.

14.8. Bibliografía de apoyo

- ***Tracepoints***: "Debugging with GDB: The GNU Source-Level Debugger"by Richard M. Stallman, Roland Pesch, Stan Shebs pp. 89-99