Some advanced debugging techniques in C under Linux

Filip Rooms

Abstract

In this short text, we will briefly discuss some techniques and tools to track down and eliminate errors in C programs.

Key words: Electric Fence, GDB, DDD, Valgrind, debugging, C, Linux.

1 GDB

GDB, the GNU Project debugger offers you the possibility to check what happens "inside" a certain program when it's being executed – or what a program was doing at the moment when it crashed.

GDB can help you in four ways to find bugs in programs:

- it can start your program and specify the circumstances that can influence your program.
- it can stop your program under given conditions.
- it can investigate what happened at the moment your program was terminated.
- make changes in your program, which allows you to experiment with correcting the influence of the bug.

The program to be debugged can be written in C, C++, Pascal (of in één van vele andere talen). Make sure that the program is compiled with the GDB debugging flag activated (to compile a certain test.c with gcc with GDB activated, use *"gcc test.c -g -o testprog"*. Here, *"-g"* activates the GNU debugger, and *"-o"* allows you to give a name to the compiled program (instead of the default name *a.out*).

`http://www.gnu.org/software/gdb/gdb.html`

2 ELECTRIC FENCE

Electric Fence is a C-library for malloc() debugging (so it needs to be linked at compilation time), which uses the virtual memory (VM) hardware of the system to check if and when a program exceeds the borders of a malloc() buffer. At the borders of such a buffer, a red zone is added. When the program enters this zone, it is terminated immediately. The library can also detect when the program tries to access memory that has already been released using free(). Because Electric Fence uses the VM hardware to detect errors, the program will be stopped at the first instruction which causes a certain buffer to be exceeded. Therefore, it becomes trivial to identify the instruction that caused the error with a debugger.

The Electric Fence library is a useful tool to detect memory errors, but when the memory errors are fixed, it is best to recompile the program without this library, because it has some unexpected side effects, like that a program compiled with Electric Fence can not migrate on a Mosix-cluster.

`http://perens.com/FreeSoftware/`

`http://www.gnu.org/directory/All_Packages_in_Directory/ElectricFence.html`

3 DDD

GNU DDD is a graphical front-end for command-line debuggers like GDB, DBX, WDB, Ladebug, JDB, XDB, the Perl debugger or the Python debugger. Next to the "usual" front-end features like allowing to watch the source code text, DDD became famous because of its interactive grafical data display, in which data structures can be shown as graphs.

We now will give an example of the plot possibilities of DDD using an example program (just to avoid confusion: there are no errors in this program). Lets have a look at example program the val1.c:

Listing 1. val1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>


int main (int argc, char **argv)
{
  int i;
  double *histo;
  histo = malloc(sizeof(double) * 100);

  for (i = 0; i < 100; i++)
    {
      double t = 10.0 * ((double) (i)) / 100.0;
      histo[i] = sin(t);
    }

  printf("End\n");
  return 1;
}
```

When we compile this program with

```
gcc val1.c -lm -g -lefence -Wall -o val1
```

and we open the executable program, *val1* in DDD. Then, we can see that an example array of 100 elements is created, in which the element in position i is given by the sine function of the index element. We first place a breakpoint on line 18 by selecting line 18 and clicking on the "stop" traffic sign icon in the top menu. If we type

```
graph display histo[0]@99
```

in the command window below, the array with elements 0 to 99 appears in the data window of DDD. When we select this array and click on the "plot" icon in the top menu, we get a graph of these array elements (fig. 2).

```
http://www.gnu.org/software/ddd/
```

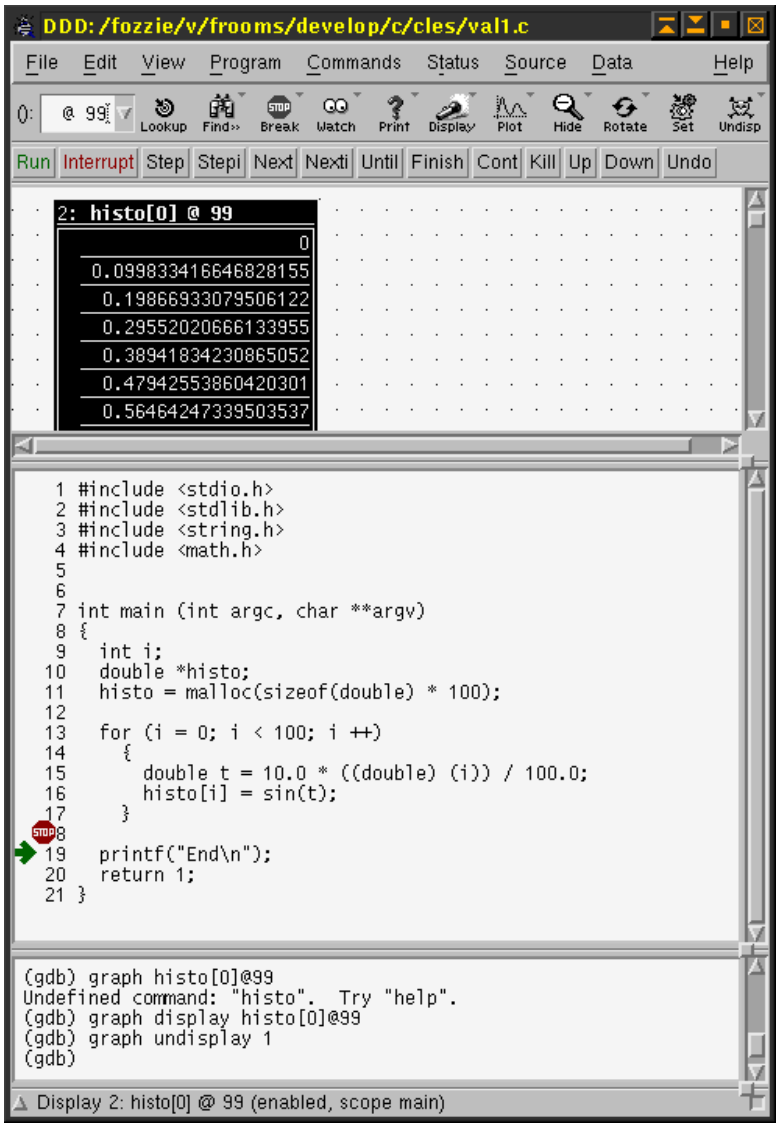

Figure 1. DDD, the Data Display Debugger with our example program *val1.c*.

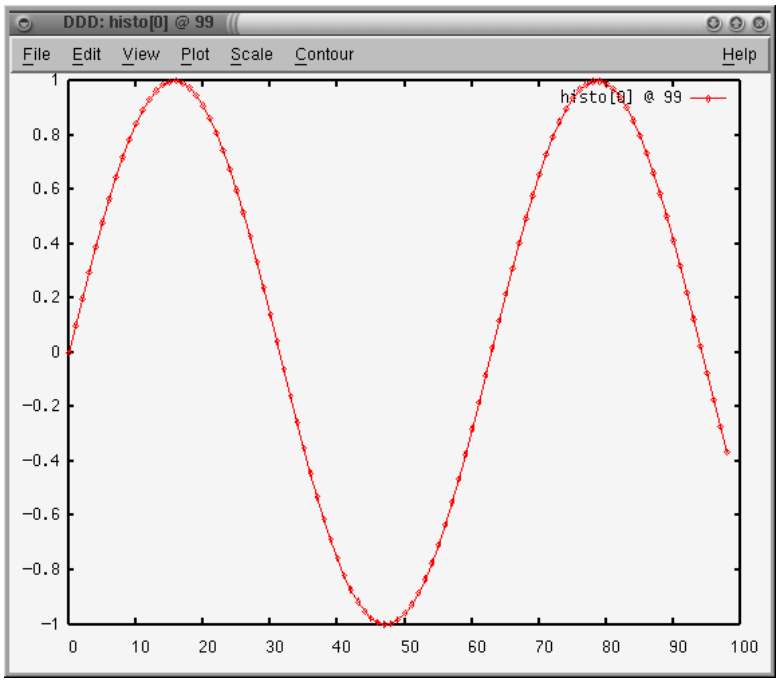

Figure 2. A graph of array elements with DDD.

4 ASK IGOR

The authors of DDD have developed another debug tool called "AskIgor". This is a webbased client, which can be found at:

`http://www.askigor.org/`

Let's take the example that the authors mention on their website:

```
/* sample.c -- Sample C program to be debugged with DDD */

#include <stdio.h>
#include <stdlib.h>

static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

int main(int argc, char *argv[])
{
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```

This program reads some numbers from standard input (at the command line) and prints them sorted at the standard output. We compile this program with:

```
gcc -g -o sample sample.c
```

A sample run of the program with as input three numbers gives:

```
$ sample 9 7 8
$ 7 8 9
```

Figure 3. Welcome to Ask Igor.

However, a sample run of the program with as input two numbers gives:

```
$ sample 9 7
$ 0 7
```

clearly an error! This is a job for Igor (named after the slave of Frankenstein, because he too got to do all the boring work). We go to the website (as shown in Figure 3). Here, we can upload the executable Linux program. Further, we have to enter with which arguments the error occurs and with which arguments the error doesn't occur. We can also upload other files that might be needed to run the program. After some time (2 minutes for the example program), Igor gives the analysis of the two runs of the program, as shown in Figure 4.

A more detailed analysis produces something like Figure 5. However, Figure 4 already gives enough information: on line 32, an array is filled with `argc - 1` elements. On line 35, this array is passed to the sorting routine with size mentioned to be `argc`. When the program was executed with two numbers as input, in fact three numbers were sorted. However, the last of these numbers was uninitialised, and contained the value 0. Because of the sorting, this number was put as first number.

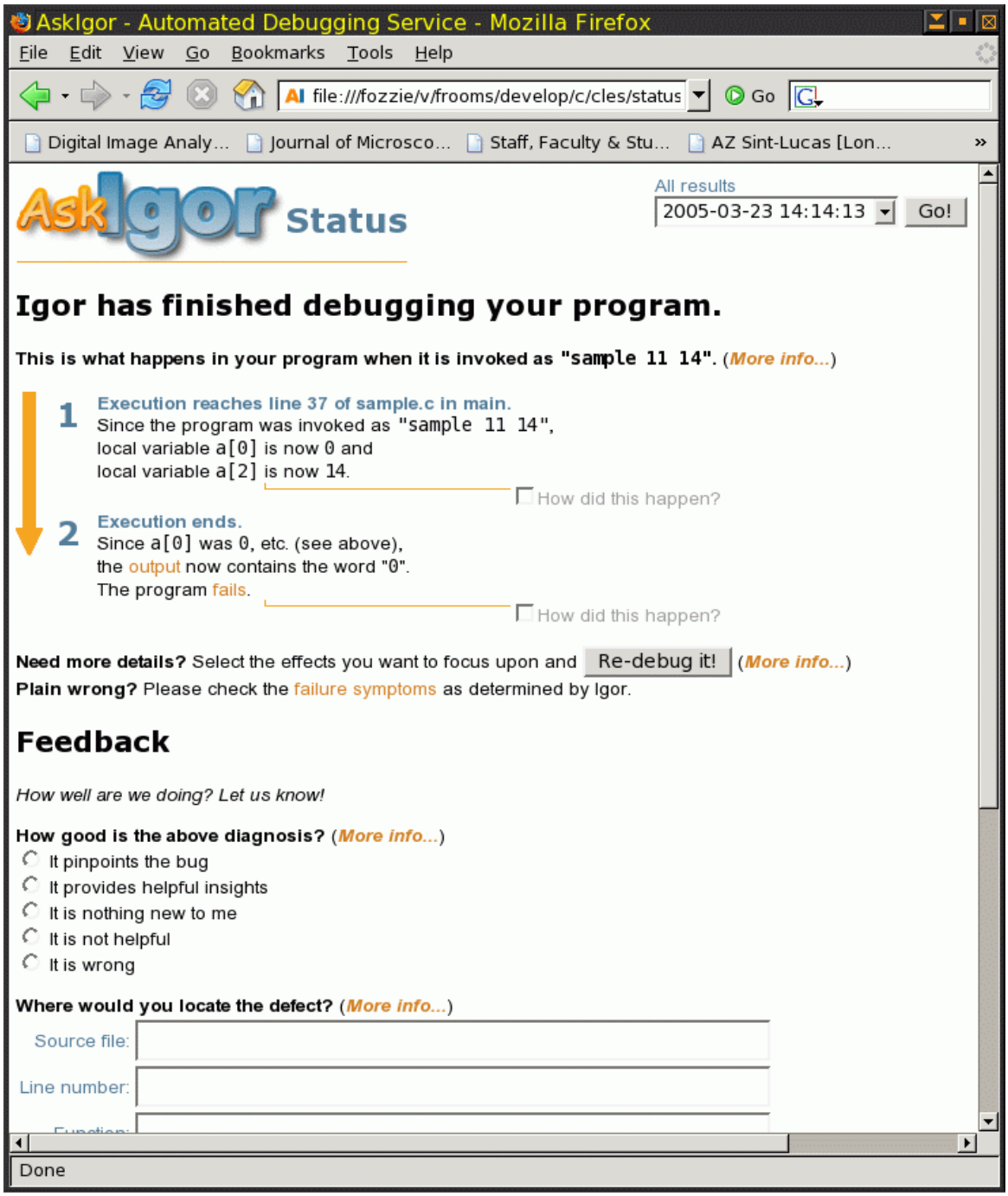

Figure 4. Analysis of the program.

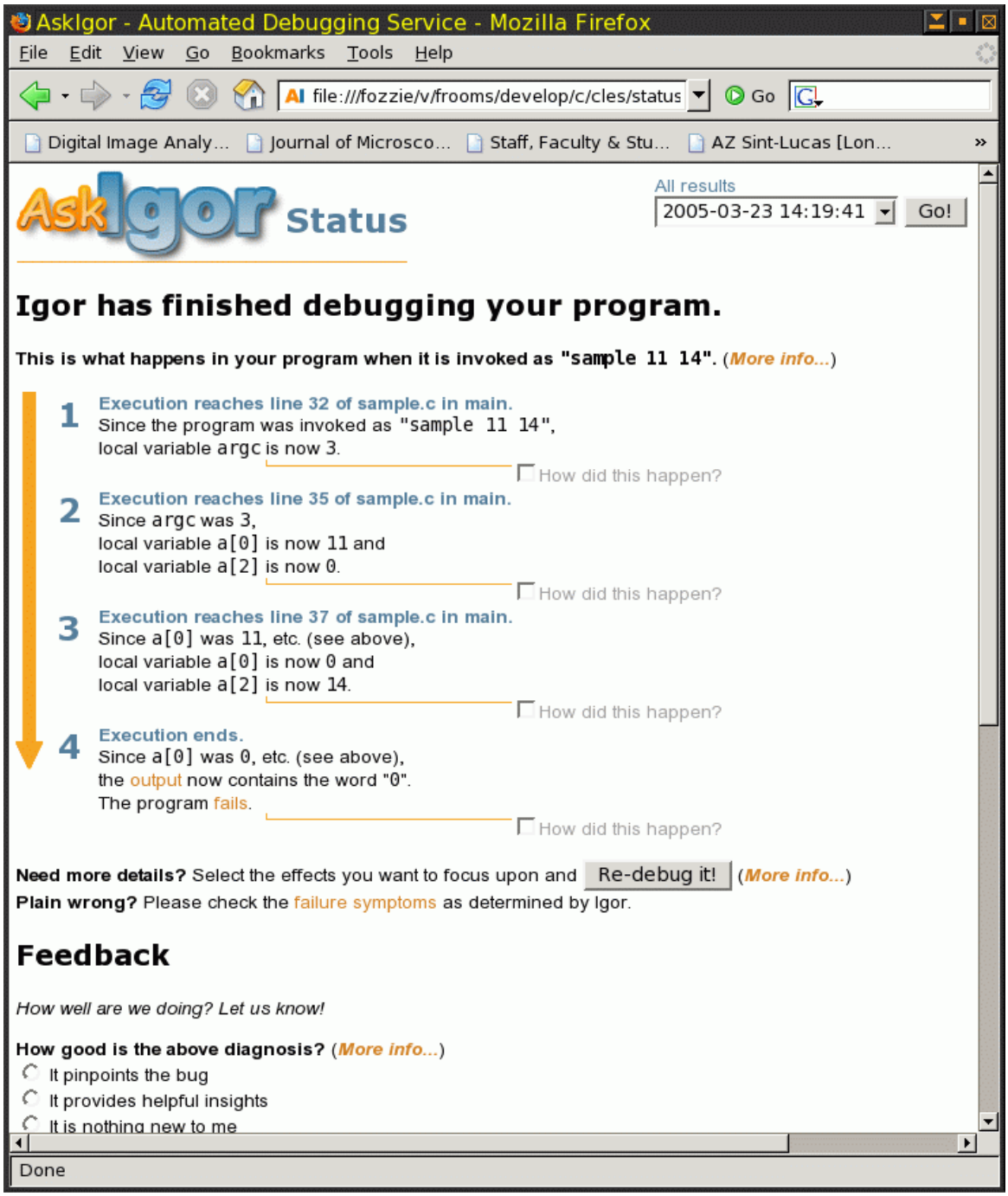

Figure 5. Detailed analysis of the program.

5 VALGRIND

Valgrind finds memory-management problems by checking every readings from and writings to memory, and intercepting and checking all calls to malloc/new/free/delete. Therefore, Valgrind can detect problems, like the usage of uninitialized memory, reading from and writing to memory that has been freed, reading and writing beyond the borders of allocated blocks, reading and writing on positions in the stack where it's not allowed, memory leaks and passing of uninitialized and/or parts of the memory not accessible by system calls. Valgrind tracks every byte the memory with nine status bits: one tracks the accessibility and the other eight if the content is valid. As a consequence, Valgrind can detect uninitialized and doesn't report false errors on bitfield operations. Valgrind can debug almost all dynamically linked ELF x86 executables (Executable and Linking Format: standard file format for executable programs in a Linux system) without any need for modification or recompilation.

A small disadvantage of valgrind is that the execution speed of the program is slowed down with a factor ranging from 30 to 50.

`http://developer.kde.org/~sewardj/`

`http://www.gnu.org/directory/valgrind.html`

This tool has also some cache profiling aspects. In order to use those, you must type *cachegrind* instead of *valgrind*. For this tool to analyse the performance of your program, there is also a graphical visualisation tool *kcachegrind*. This program facilitates the interpretation of the numbers that *cachegrind* generates.

6 SOME EXAMPLE PROGRAMS

6.1 Memory error

Listing 2. val2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <assert.h>

int main (int argc, char **argv)
{
  int i;
  double *histo;

  histo = malloc(sizeof(double) * 60);

  for (i = 0; i < 100; i++)
    {
      histo[i] = i * i;
    }

  printf("End\n");

  return 1;
}
```

Let's have a look at example program *val2.c*, where a memory error is obvious (but we pretend we didn't notice). We compile this program with:

```
gcc val2.c -g -lefence -o val2 -Wall
```

with *gcc* the GNU C compiler. We see that we have to give some extra flags to the compiler. With the option *-Wall* we get warnings which already points us to some errors, like variables that are used uninitialized,

or statements with no effect. It is recommended always to use this option. The *-g* flag will take care that the compiler includes debug infomation for the GNU debugger (GDB). The last flag *-lefence* is needed to link the library Electric Fence to our program.

In program *val2*, an array of 60 elements is created, which we now try to fill until element 100. OK, let's see which errors we find with DDD. We open *val2* in DDD, and let it run by giving as input in the command window (below) *run*, with the arguments that the programs normally requires (in this case, none). The program *val2*, first creates the array and tries to fill it then up to element 100. The output of the program looks like this:

```
Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

Program received signal SIGSEGV, Segmentation fault.
0x080484da in main (argc=1, argv=0xbffff5d4) at val2.c:17
```

and grafically in the screen of DDD (shown in in figure 6). In this figure, the red arrow on line 17 indicates the

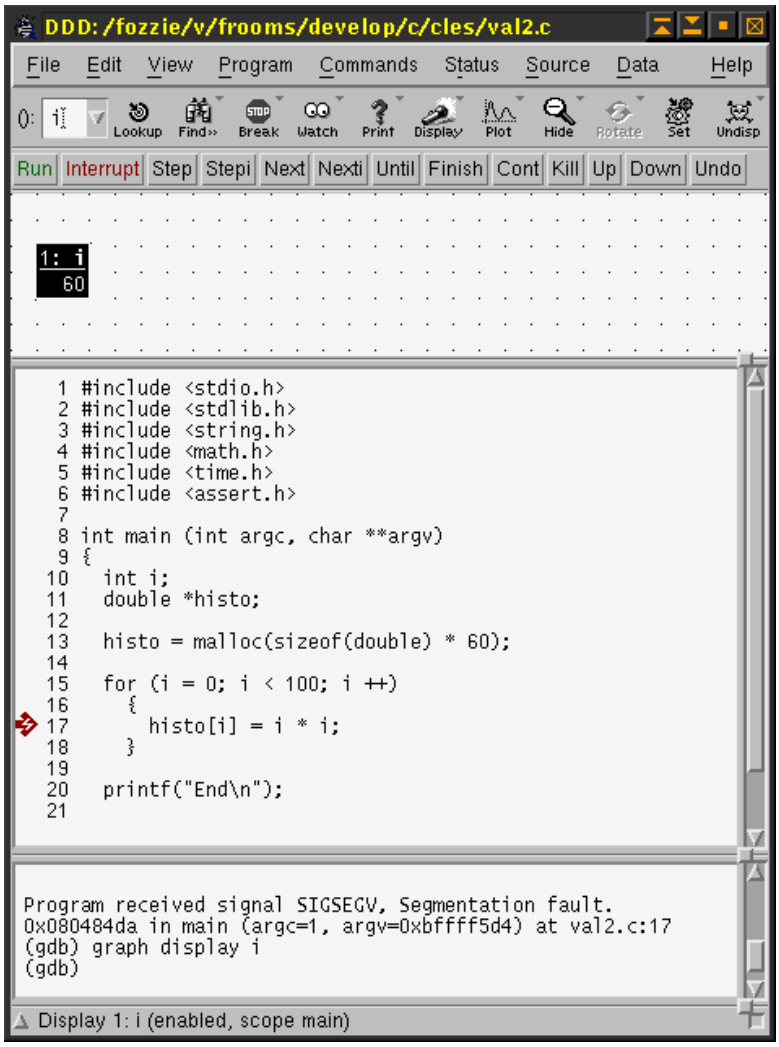

Figure 6. Error detection with DDD.

place where the error occurred. When we check the index i of the array *histo* by double clicking on the variable i in the source code window of DDD, we see in the data window (top part of the DDD window, with the grid) that the value of i is 60. This is right, because by linking with *-lefence* during the compilation, the program crashes exactly on that moment when the program tries to read or write where it is not allowed. On positions 0 to 59, we could write, so no problem. At position 60, the program crashes, and DDD indicates neatly with a red arrow on which line the error occured.

We also run valgrind with default flags on *valtest*:

```
valgrind val2
```

It is better however to give some extra argument flags, like:

```
valgrind --gdb-attach=yes --error-limit=no --leak-check=yes val2
```

Flag 1 indicates that we want to use GDB when an error is found by valgrind (this allows us to check values of variable in the program at the moment the error occurred). Flag 2 makes sure that valgrind completes the check (default, Valgrind stops when the program contains more than a certain number of errors), and flag 3 allows us to check the memory usage (e.g., when we allocate a temporary array in a function without freeing it, we cause the program to consume more and more memory).

With these options, the output of valgrind then looks like this:

```
==6455== Memcheck, a.k.a. Valgrind, a memory error detector for x86-linux.
==6455== Copyright (C) 2002-2003, and GNU GPL'd, by Julian Seward.
==6455== Using valgrind-2.0.0, a program supervision framework for x86-linux.
==6455== Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
==6455== Estimated CPU clock rate is 1814 MHz
==6455== For more details, rerun with: -v
==6455==
==6455== Invalid write of size 8
==6455==    at 0x80484DA: main (val2.c:17)
==6455==    by 0x4026A081: __libc_start_main (in /lib/i686/libc-2.2.5.so)
==6455==    by 0x80483F0: (within /fozzie/v/frooms/develop/c/cles/val2)
==6455==    Address 0x411B0204 is 0 bytes after a block of size 480 alloc'd
==6455==    at 0x400260DC: malloc (vg_replace_malloc.c:153)
==6455==    by 0x80484A8: main (val2.c:13)
==6455==    by 0x4026A081: __libc_start_main (in /lib/i686/libc-2.2.5.so)
==6455==    by 0x80483F0: (within /fozzie/v/frooms/develop/c/cles/val2)
==6455==
==6455== ---- Attach to GDB ? --- [Return/N/n/Y/y/C/c] ----
```

Here, something goes wrong, so we switch to GDB. With the command *up*, we walk through the functions which called each other at the moment of the error, until we reach a reference from our own source code.

```
==6455== starting GDB with cmd: /usr/bin/gdb -nw /proc/6455/exe 6455
GNU gdb 5.2.1-2mdk (Mandrake Linux)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linux-gnu"...
Attaching to program: /fozzie/v/frooms/develop/c/cles/val2, process 6455
Reading symbols from /v/frooms/bin/val2//lib/valgrind/vgskin_memcheck.so...
done.
Loaded symbols for /v/frooms/bin/val2//lib/valgrind/vgskin_memcheck.so
Reading symbols from /v/frooms/bin/val2//lib/valgrind/valgrind.so...done.
Loaded symbols for /v/frooms/bin/val2//lib/valgrind/valgrind.so
Reading symbols from /usr/lib/libefence.so.0...done.
Loaded symbols for /usr/lib/libefence.so.0
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /v/frooms/bin/val2//lib/valgrind/libpthread.so.0...done.
Loaded symbols for /v/frooms/bin/val2//lib/valgrind/libpthread.so.0
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
vg_do_syscall3 (syscallno=4294966784, arg1=6500, arg2=0, arg3=0)
    at vg_mylibc.c:92
92      }
(gdb)
```

Here, we can use GDB commands to track the error. Since the message here is not really clear, we want to go up the stack to see which instruction from our program caused the mistake. So here, we type *up*, after which we see

```
(gdb) up
#1  0x00001964 in ?? ()
(gdb)
```

Not really clear yet, so we go through the stack level by level with the command *up*

```
(gdb) up
#2  0x4017fedf in vgPlain_start_GDB_whilst_on_client_stack () at vg_main.c:1816
1816       res = VG_(system)(buf);
(gdb) up
#3  0x40186618 in vgPlain_swizzle_esp_then_start_GDB ()
   from /v/frooms/bin/val2//lib/valgrind/valgrind.so
(gdb) up
#4  0x080484da in main (argc=1, argv=0xbffff554) at val2.c:17
17            histo[i] = i * i;
(gdb)
```

until our memory error is found again on line 17 (as indicated at the beginning of the output line of Valgrind... With the GDB command *display i* we find

```
(gdb) display i
1: i = 60
```

6.2 Error by forgetting to initialize

Listing 3. val3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

int main (int argc, char **argv)
{
  double interval;
  double k, l;
  interval = atof(argv[1]);

  if (interval == 0.1) {k = 3.14;}
  if (interval == 0.2) {k = 2.71;}

  l = 5.0 * exp(k);

  printf("l = %lf\n", l);
  return 1;
}
```

In the program above, there is a subtle error. We compile with

```
gcc val3.c -lm -g -lefence -o val3
```

This error doesn't cause the program to crash, so you can run the program without noticing an error. The problem is this: the user has to give an argument as input. However, if this input value is not equal to 0.1 or 0.2, the value for k is not initialized, and we may get unexpected results. So we examine our program with Valgrind, and give as argument to the program 0.9

```
valgrind --gdb-attach=yes --error-limit=no --leak-check=yes val3 0.9
```

The result looks like this:

```
==5531== Memcheck, a.k.a. Valgrind, a memory error detector for x86-linux.
==5531== Copyright (C) 2002-2003, and GNU GPL'd, by Julian Seward.
==5531== Using valgrind-2.0.0, a program supervision framework for x86-linux.
==5531== Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
==5531== Estimated CPU clock rate is 1810 MHz
==5531== For more details, rerun with: -v
==5531==
==5531== Use of uninitialised value of size 8
==5531==    at 0x402570BA: exp (in /lib/i686/libm-2.2.5.so)
==5531==    by 0x8048479: main (val3.c:15)
==5531==    by 0x40289081: __libc_start_main (in /lib/i686/libc-2.2.5.so)
==5531==    by 0x8048350: (within /fozzie/v/frooms/develop/c/cles/val3)
==5531==
==5531== ---- Attach to GDB ? --- [Return/N/n/Y/y/C/c] ----
```

Here Valgrind detects that something is wrong and says: *"Use of uninitialised value of size 8"*. We don't know where this uninitialised value occurs, so we confirm that we want to switch to GDB, and go back through the stack with *up* until we can see which instruction in our code caused the error.

```
==5531== starting GDB with cmd: /usr/bin/gdb -nw /proc/5531/exe 5531
GNU gdb 5.2.1-2mdk (Mandrake Linux)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linux-gnu"...
Attaching to program: /fozzie/v/frooms/develop/c/cles/val3, process 5531
Reading symbols from /v/frooms/bin/val2//lib/valgrind/vgskin_memcheck.so...
done.
Loaded symbols for /v/frooms/bin/val2//lib/valgrind/vgskin_memcheck.so
Reading symbols from /v/frooms/bin/val2//lib/valgrind/valgrind.so...done.
Loaded symbols for /v/frooms/bin/val2//lib/valgrind/valgrind.so
Reading symbols from /lib/i686/libm.so.6...done.
Loaded symbols for /lib/i686/libm.so.6
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
vg_do_syscall3 (syscallno=4294966784, arg1=5623, arg2=0, arg3=0)
    at vg_mylibc.c:92
92      }
(gdb)
```

Again, we go up the stack with *up* to the point where the error occured in our program:

```
(gdb) up
#1  0x000015f7 in ?? ()
(gdb) up
#2  0x4017fedf in vgPlain_start_GDB_whilst_on_client_stack () at vg_main.c:1816
1816       res = VG_(system)(buf);
(gdb) up
#3  0x40186618 in vgPlain_swizzle_esp_then_start_GDB ()
   from /v/frooms/bin/val2//lib/valgrind/valgrind.so
(gdb) up
#4  0x402570ba in exp () from /lib/i686/libm.so.6
(gdb) up
#5  0x0804847a in main (argc=2, argv=0xbffff544) at val3.c:15
15        l = 5.0 * exp(k);
```

Here we can see what the problem is: the program computes l, which depends on a variable k that hasn't been initialized. We gave a value to the program as an argument, but only if this value is equal to 0.1 or 0.2, a value is assigned to k. In our case, k is used uninitialized to calculate l.

6.3 Tracking memory leaks

Listing 4. val4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
int fun1()
  {
    double *sss;
    sss = malloc(1000);
  }


int main (int argc, char **argv)
{
  double k;

  for (k = 0; k < 100; k++)
    {
       fun1();
    }


  return 1;
}
```

In the example above, a memory block is allocated in a function, but never freed again. This function is called iteratively, which causes more and more memory to be allocated without being used.

We compile with

```
gcc val4.c -g -Wall -o val4
```

and check what valgrind will tell us about this program with the instruction

```
valgrind --gdb-attach=yes --leak-check=yes  val4
```

The output of valgrind looks like this:

```
==30250== valgrind-1.0.2, a memory error detector for x86 GNU/Linux.
==30250== Copyright (C) 2000-2002, and GNU GPL'd, by Julian Seward.
==30250== Estimated CPU clock rate is 1807 MHz
==30250== For more details, rerun with: -v
==30250==
==30250==
==30250== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==30250== malloc/free: in use at exit: 100000 bytes in 100 blocks.
==30250== malloc/free: 100 allocs, 0 frees, 100000 bytes allocated.
==30250== For counts of detected errors, rerun with: -v
==30250== searching for pointers to 100 not-freed blocks.
==30250== checked 3464616 bytes.
==30250==
==30250== definitely lost: 100000 bytes in 100 blocks.
==30250== possibly lost:   0 bytes in 0 blocks.
==30250== still reachable: 0 bytes in 0 blocks.
```

```
==30250==
==30250== 100000 bytes in 100 blocks are definitely lost in loss record 1 of 1
==30250==    at 0x4003B575: malloc (in /usr/lib/valgrind/valgrind.so)
==30250==    by 0x804834F: fun1 (in /fozzie/v/frooms/develop/c/cles/val4)
==30250==    by 0x804838F: main (val4.c:18)
==30250==    by 0x40270082: (within /lib/i686/libc-2.2.5.so)
==30250==
==30250== LEAK SUMMARY:
==30250==    definitely lost: 100000 bytes in 100 blocks.
==30250==    possibly lost:   0 bytes in 0 blocks.
==30250==    still reachable: 0 bytes in 0 blocks.
==30250== Reachable blocks (those to which a pointer was found) are not shown.
==30250== To see them, rerun with: --show-reachable=yes
==30250==
```

from which we can conclude that in the function *fun1* memory is allocated, but it is never released later.

7 CONCLUSION

In this tutorial, we have discussed some useful tools for debugging C programs under Linux. Further, we also demostrated these tools on some example programs with errors like memory reading/writing to memory that was not allocated, detecting the usage of uninitialized variables in programs and detecting memory leaks.

I would like to thank Leyden Martinez for proof-reading this text. Further suggestions, remarks, ... can be mailed to *filip.rooms@gmail.com*.