Enkele geavanceerde debugtechnieken in C voor linuxprogramma's

Filip Rooms

Abstract

In dit artikeltje willen we kort enkele technieken en hulpmiddelen bespreken om fouten uit C-programma's te kunnen halen.

Sleutelwoorden: *Electric Fence, GDB, DDD, Valgrind.*

1 GDB

GDB, de GNU Project debugger geeft je de mogelijkheid om te zien wat er in feite gebeurt "aan de binnenkant" van een bepaald programma wanneer het wordt uitgevoerd – of wat een programma aan het doen was op het moment dat het crashte.

GDB kan in hoofdzaak vier dingen doen die je helpen om bugs in programma's te vinden terwijl ze zich voordoen:

- opstarten van uw programma and specifiëren van omstandigheden die het gedrag van het programma kunnen beïnvloeden;
- stoppen van uw programma onder bepaalde voorwaarden;
- onderzoeken wat er gebeurde op het moment dat uw programma beëindigd werd;
- veranderingen in uw prgramma aanbrengen zodat je kan experimenteren met het corrigeren van de invloed van een bug.

Het te debuggen programma kan geschreven zijn in C, C++, Pascal (of in één van vele andere talen).

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

2 ELECTRIC FENCE

Electric Fence is een malloc() debugger, die de virtual memory hardware van het systeem gebruikt om op te sporen wanneer een programma de grenzen van een malloc() buffer overschrijdt. Aan de grenzen van zo'n buffer wordt een rode zone afgebakend. Als het programma zich daarin begeeft, wordt het onmiddellijk beëindigd. De bibliotheek kan ook toegang tot reeds met free() vrijgegeven geheugen detecteren. Omdat Electric Fence gebruik maakt van de VM hardware om fouten te detecteren, zal het programma gestopt worden bij de eerste instructie die ervoor zorgt dat een bepaalde buffer overschreden wordt. Daardoor wordt het triviaal om met een debugger de instructie die in de fout gaat aan het licht te brengen.

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

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

3 DDD

GNU DDD is een grafische front-end voor command-line debuggers zoals GDB, DBX, WDB, Ladebug, JDB, XDB, de Perl debugger of de Python debugger. Naast de "gebruikelijke" front-end features zoals het bekijken van de broncode tekst, is DDD bekend geworden door zijn interaktieve grafische data display, waarbij data structuren getoond kunnen worden als grafieken.

We geven nu een voorbeeld van de plotmogelijkheden van DDD aan de hand van een voorbeeldprogramma (voor alle duidelijkheid: er zitten geen errors in dit programma). Bekijken we volgend voorbeeldprogramma valtest2.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;
}
```

Als we dit voorbeeldprogramma compileren met

```
gcc valtest2.c -lm -g -lefence -Wall -o val2
```

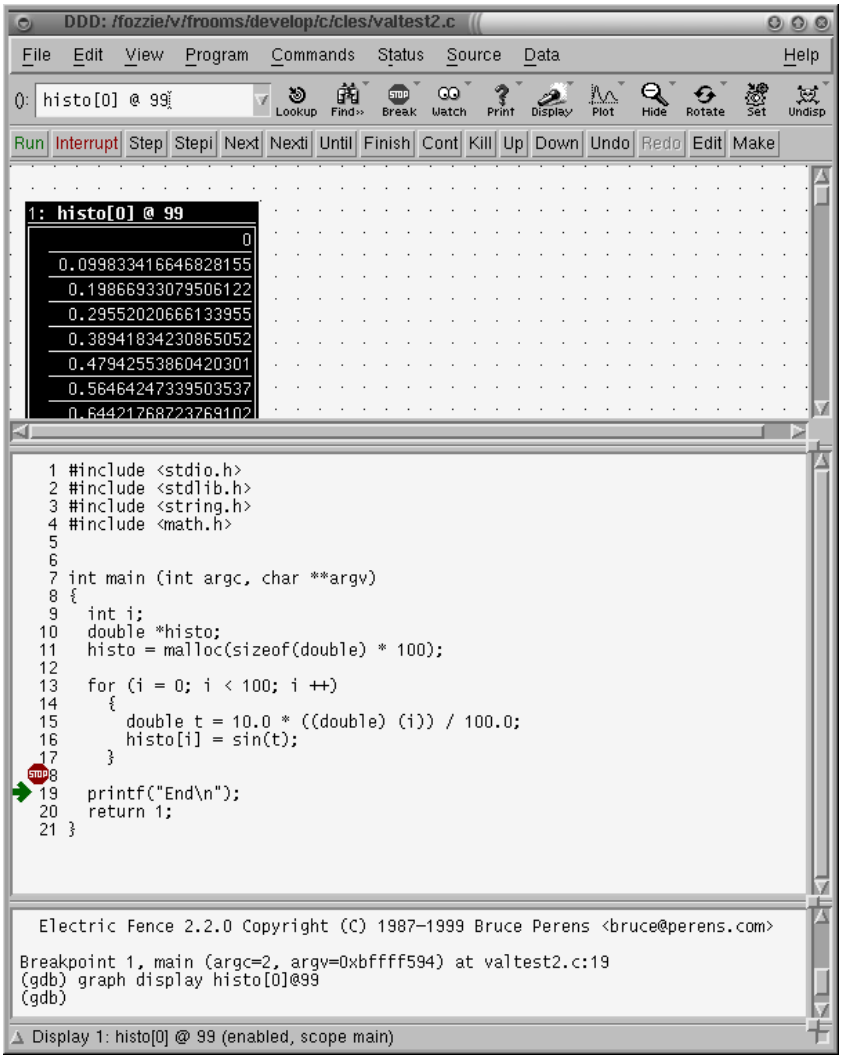
en we openen *val2* in DDD, krijgen we een scherm als in Figuur 1.

We zien dan dat er een voorbeeldarray van 100 elementen wordt aangemaakt, waarbij het element op positie i wordt gegeven als een sinusfunctie van het indexelement. We plaatsen eerst een breakpoint op lijn 18 door op lijn 18 te selecteren en dan op het "stop" verkeersbordje in het menu bovenaan te klikken. Geven we nu

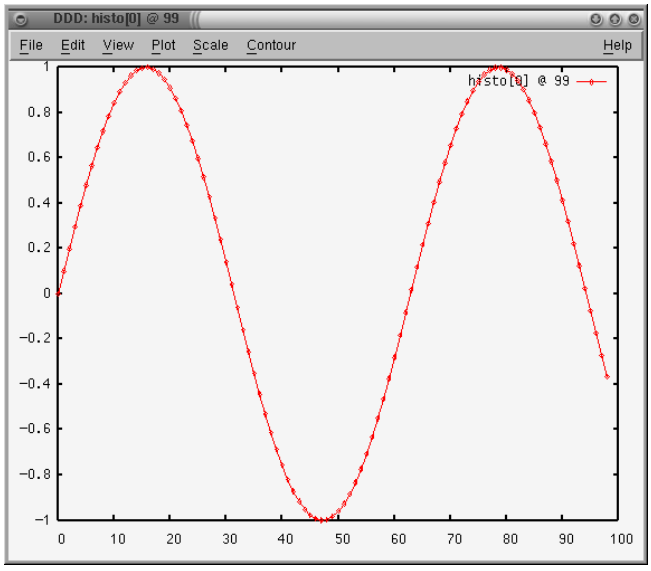
```
graph display histo[0]@99
```

in het commadovenster onderaan, dan verschijnt bovenaan de array met de elementen 0 tot 99. Wanneer we die selecteren en dan op "plot" klikken, krijgen we een grafiek van deze array-elementen (Figuur 2).

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



Figuur 1. DDD, de Data Display Debugger in aktie op het voorbeeldprogramma.



Figuur 2. Een grafiek van een array met DDD.

4 ASK IGOR

De auteurs van DDD hebben een nieuw debug-toolke ontwikkeld, namelijk "AskIgor", een webgebaseerde client die te vinden is op:

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

Nemen we het voorbeeld dat de auteurs vermelden op hun 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;
}
```

Dit is een programma dat een aantal getallen vanop de standaard invoer inleest, en op de standaard uitvoer deze gesorteerd weergeeft. We compileren dit programma met:

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

Als we dit programma nu eens proefdraaien:

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

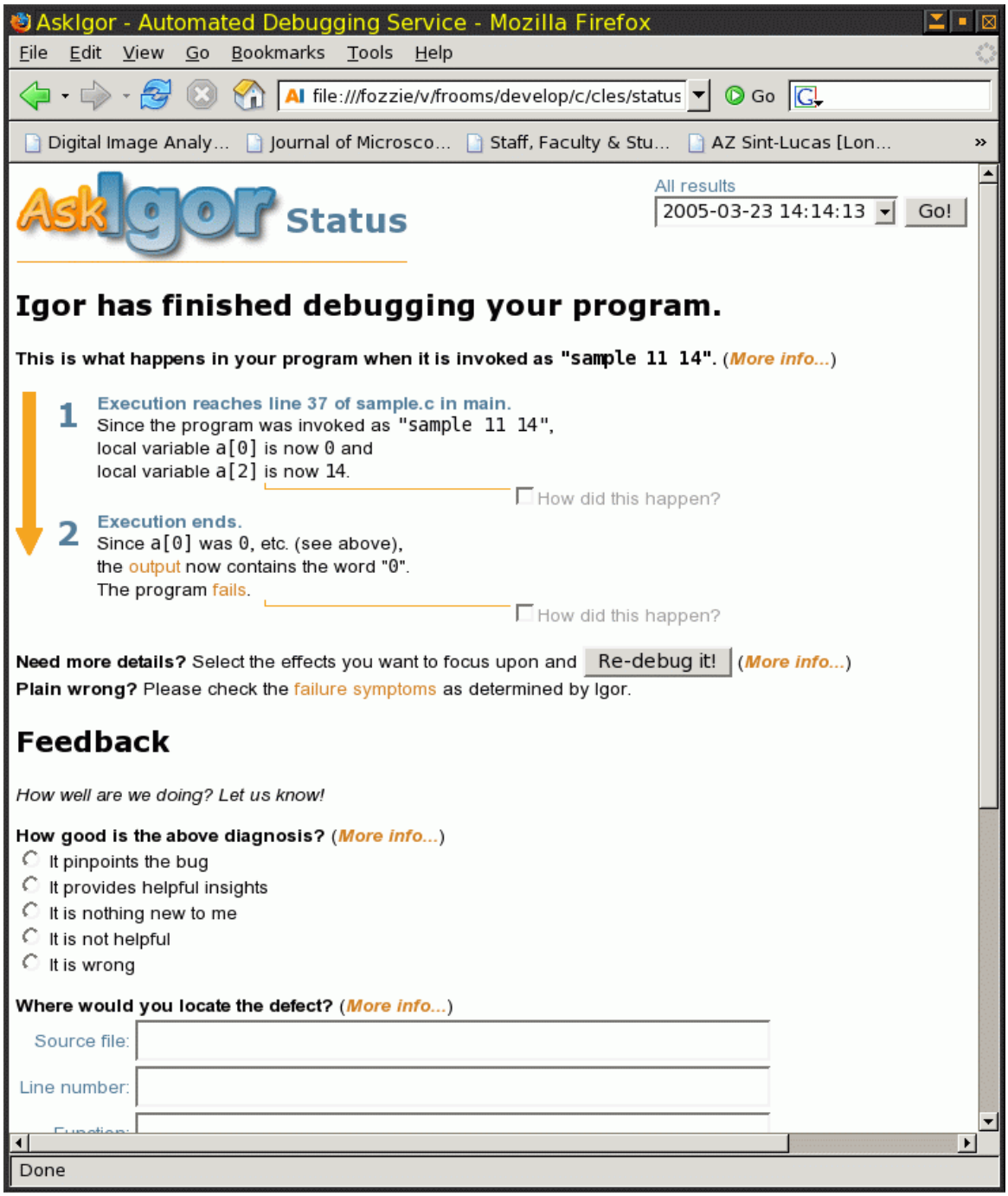
Figuur 3. Welkom bij Ask Igor.

Als we dit programma echter laten draaien met twee getallen:

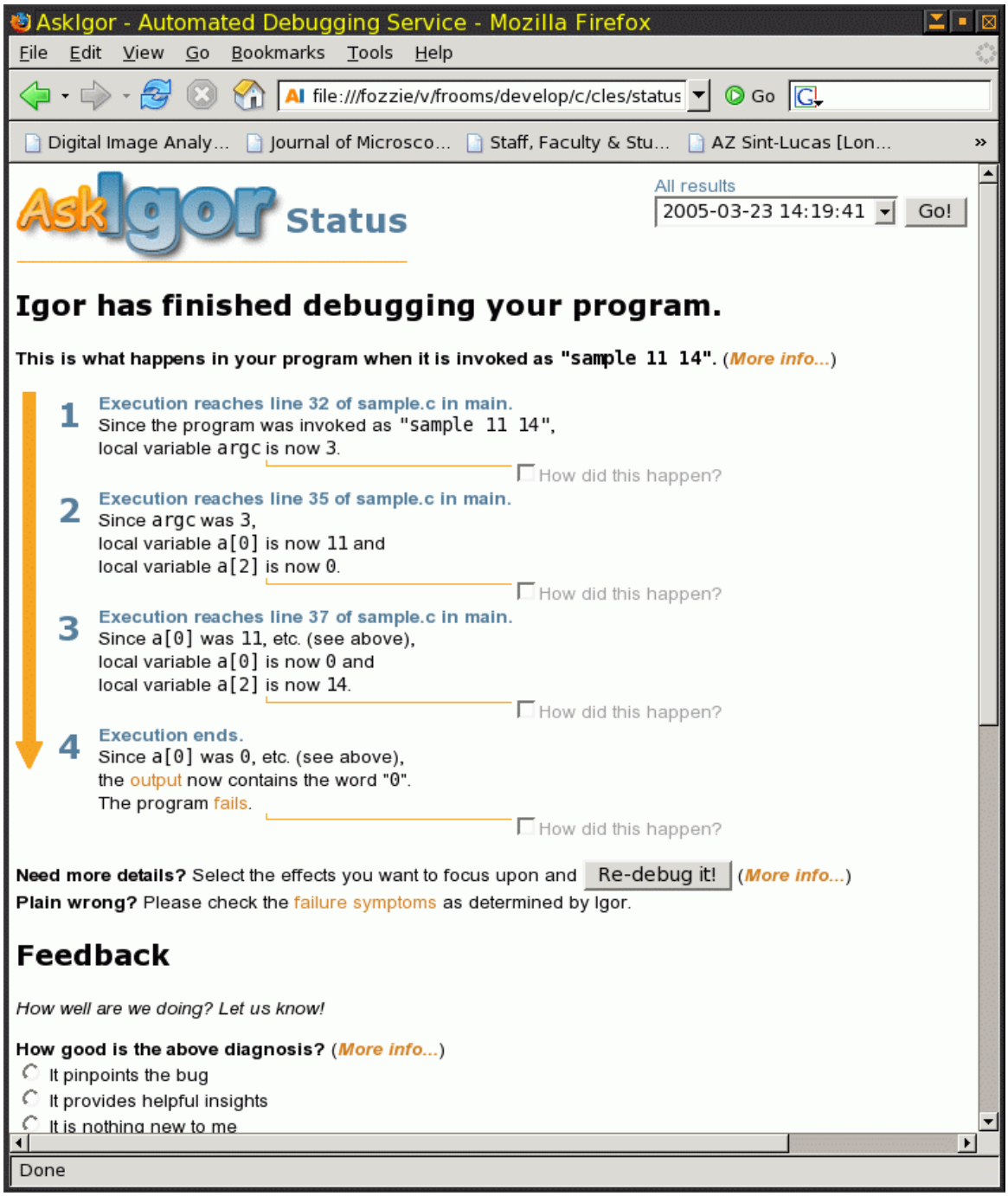
```
$ sample 9 7
$ 0 7
```

dan loopt er duidelijk iets mis. Dit is een klusje voor Igor (genoemd naar het slaafje van Frankenstein, omdat die ook alle saaie klusjes mocht opknappen). We gaan naar de website (Figuur 3), waar we de volgende pagina te zien krijgen. We kunnen daar gewoon het uitvoerbaar Linux programma uploaden, en intikken met welke argumenten er fouten optreden, en met welke argumenten niet. Ook kunnen eventuele extra files nodig voor het uitvoeren van het programma worden ge-upload. Na gemiddeld twee minuten wachten geeft Igor de analyse zoals weergegeven in Figuur 4.

Een gedetailleerdere analyse geeft dan iets zoals in Figuur 5. Figuur 4 geeft ons echter al genoeg informatie: op lijn 32 wordt een array gevuld met `argc - 1` elementen. Op lijn 35 wordt deze array echter aan de sorteerroutine doorgegeven met als grootte `argc`. Wanneer het programma werd uitgevoerd met twee getallen als argumenten, werden er in feite drie elementen gesorteerd. Het laatste element was echter niet geïnitialiseerd, en stond op 0, wat door het sorteren vooraan is beland.



Figuur 4. Analyse van het programma.



Figuur 5. Gedetailleerdere analyse van het programma.

5 VALGRIND

Valgrind vindt memory-management problemen door alle uitlezen van en schrijven naar het geheugen te controleren, en alle aanroepen van malloc/new/free/delete te onderscheppen en te controleren. Daardoor kan Valgrind problemen detecteren zoals het gebruik van ongeïnitializeerd geheugen, uitlezen van en schrijven naar geheugen dat reeds werd vrijgegeven, lezen/schrijven voorbij de uiteinden van malloc'd blokken, lezen/schrijven van plaatsen in de stack waar het niet mag, memory leaks en doorgeven van ongeïnitializeerde en/of niet-addresseerbare delen van het geheugen aan system calls. Valgrind volgt elke byte van het geheugen met negenn status bits: één volgt de adresseerbaarheid en de andere acht de geldigheid. Als resultaat daarvan kan Valgrind het gebruik van enkele ongeïnitializeerde bits detecteren en rapporteert het geen valse errors over bitfield operaties. Valgrind debugt bijna elke dynamisch-gelinkte ELF x86 executable zonder dat die moet aangepast of gehercomplieerd moet worden.

Klein nadeel van valgrind is dat de uitvoer van het programma met een factor 30 tot 50 wordt vertraagd.

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

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

Ook zijn er aan deze tool cache profiling aspecten. Daarvoor moet je *cachegrind* ipv *valgrind* intypen voor het aanroepen van het programma. Voor deze tool voor de analyse van de performantie van uw programma bestaat er ook een grafische visualisatietool *kcachegrind*. Deze laat toe de massa getallen die *cachegrind* genereert beter te interpreteren.

6 ENKELE VOORBEELDPROGRAMMA'S

6.1 Geheugenfout

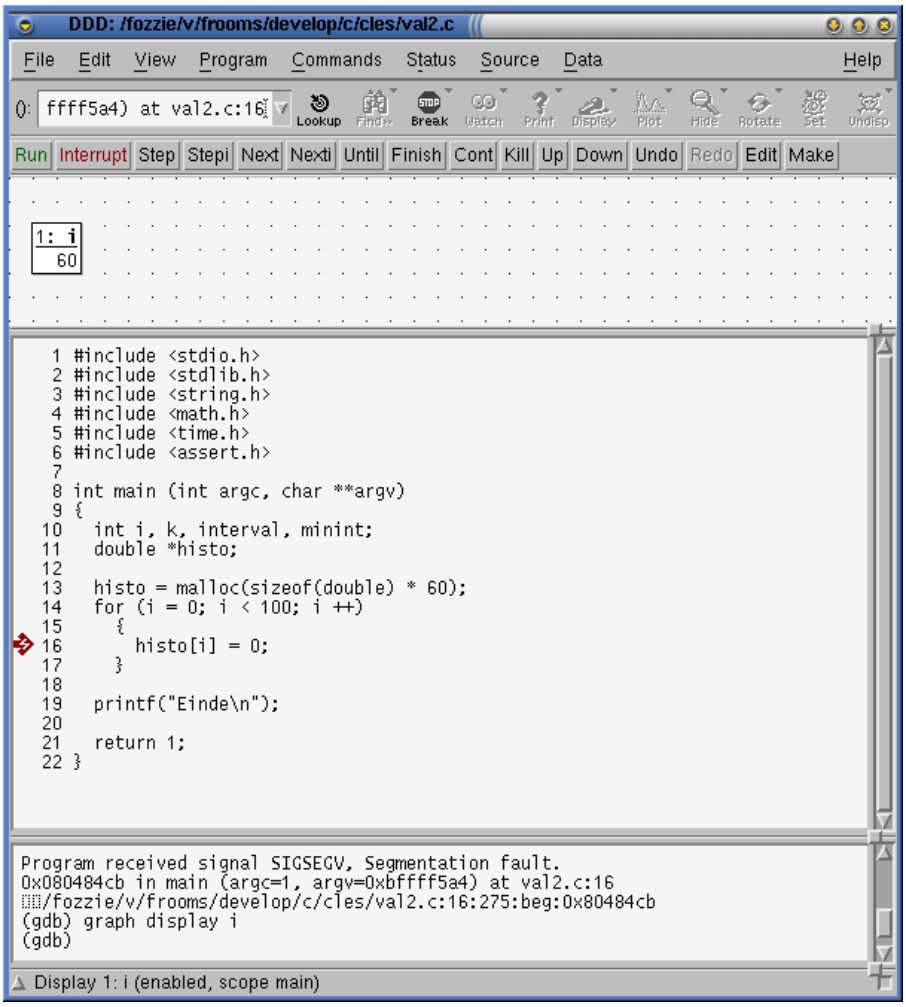
Listing 1. val2.c

```
#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;
}
```

Hierboven hebben we een voorbeeldprogramma (val2.c), waar natuurlijk een kanjer van een geheugenfout zit (maar dat weten we zogezegd niet). We compileren dit programma met:

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

waarbij *gcc* de GNU C compiler is. We zien dat aan de compiler enkele extra compileropties meegegeven worden. Met de optie *-Wall* krijgen we alle mogelijk waarschuwingen voor eventuele fouten. Deze optie onderschept al een aantal mogelijke fouten, en het is aangeraden deze altijd te gebruiken. De optie *-g* zorgt ervoor



Figuur 6. Errordetectie met DDD.

dat er gecompileerd wordt met debug-informatie voor de GNU debugger(GDB). De laatste optie *-lefence* zorgt ervoor dat er gelinkt wordt aan de bibliotheek Electric Fence.

In programma *val2* wordt een array aangemaakt van 60 elementen, die we proberen vullen tot element 100. OK, laten we eens DDD los om te zien welke fouten die vindt. We openen *val2* in DDD, en laten het lopen door in het commandovenster (onderaan) *run 0.1* in te geven (er wordt één argument verwacht als we het programma willen laten lopen, in dit geval het getal 0.1). In programma *val2* wordt een array aangemaakt van 60 elementen, die we proberen vullen tot element 100. De uitvoer ziet er dan als volgt uit:

```
Program received signal SIGSEGV, Segmentation fault.
0x080484cb in main (argc=1, argv=0xbffff5a4) at val2.c:16
/fozzie/v/frooms/develop/c/cles/valtest.c:21:366:beg:0x8048581
```

en grafisch (Figuur 6).

waarbij het rode pijltje op regel 16 laat zien waar het misloopt. Als we dan de index i van de array *histo* visualizeren door erop te dubbelklikken, dan zien we in het datavenster (bovenste venster, met de grid) dat i de waarde 60 heeft. Dat is logisch, want door te compileren met *-lefence* wordt ervoor gezorgd dat het programma crasht precies op het moment dat er geschreven wordt op een plaats waar het niet mag. Op plaat 0 tot 59 mochten we schrijven, dus geen probleem. Op plaats 60 crasht het programmna, en een rood pijltje duidt netjes de plaats aan waar de fout optrad.

We laten ook valgrind eens los op hetzelfde programma met default vlaggen:

```
valgrind valtest 1
```

Beter is echter een aantal extra opties mee te geven, zoals:

```
valgrind --gdb-attach=yes --error-limit=no --leak-check=yes  val2 1
```

Optie 1 geeft aan dat we GDB willen aanroepen wanneer er zich een fout voordoet (waardoor we allerlei waarden van het programma kunnen opvragen op het moment dat er iets foutgaat, zie boven). Optie 2 verhindert dat valgrind er de brui aan geeft na een bepaald aantal fouten, en optie 3 laat toe het geheugengebruik te controleren (bijv. in een functie een tijdelijke array malloc()'en, zonder deze weer vrij te geven, geeft aanleiding tot nodeloos groeiend geheugengebruik van een programma).

De uitvoer van valgrind ziet er als volgt uit.

```
==1947== valgrind-1.0.2, a memory error detector for x86 GNU/Linux.
==1947== Copyright (C) 2000-2002, and GNU GPL'd, by Julian Seward.
==1947== Estimated CPU clock rate is 1805 MHz
==1947== For more details, rerun with: -v
==1947==
==1947== Invalid write of size 4
==1947==    at 0x80484C8: main (in /fozzie/v/frooms/develop/c/cles/val2)
==1947==    by 0x40297082: (within /lib/i686/libc-2.2.5.so)
==1947==    by 0x80483F1: (within /fozzie/v/frooms/develop/c/cles/val2)
==1947==    Address 0x42B59204 is 0 bytes after a block of size 480 alloc'd
==1947==    at 0x4003B575: malloc (in /usr/lib/valgrind/valgrind.so)
==1947==    by 0x80484A9: main (in /fozzie/v/frooms/develop/c/cles/val2)
==1947==    by 0x40297082: (within /lib/i686/libc-2.2.5.so)
==1947==    by 0x80483F1: (within /fozzie/v/frooms/develop/c/cles/val2)
==1947==
==1947== ---- Attach to GDB ? --- [Return/N/n/Y/y/C/c] ---- Y
```

Hier loopt iets mis, dus schakelen we weer GDB in. Met het commando *up* wandelen we doorheen de lijst met functies die elkaar hebben aangeroepen op het moment dat de fout optrad, tot we komen aan een verwijzing naar onze eigen broncode.

```
==1947== starting GDB with cmd: /usr/bin/gdb -nw /proc/1947/exe 1947
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 1947
Reading symbols from /usr/lib/valgrind/valgrind.so...done.
Loaded symbols for /usr/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/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 /usr/lib/valgrind/libpthread.so.0...done.
Loaded symbols for /usr/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=1951, arg2=0, arg3=0)
    at vg_mylibc.c:92
92      vg_mylibc.c: No such file or directory.
        in vg_mylibc.c
(gdb) up
#1  0x0000079f in ?? ()
(gdb)
#2  0x4005633d in vgPlain_start_GDB_whilst_on_client_stack () at vg_main.c:1379
1379    vg_main.c: No such file or directory.
        in vg_main.c
(gdb)
```

```
#3  0x4005e958 in vgPlain_swizzle_esp_then_start_GDB ()
   from /usr/lib/valgrind/valgrind.so
(gdb)
#4  0x080484c8 in main (argc=2, argv=0xbffff594) at val2.c:16
16            histo[i] = 0;
```

waar weer onze geheugenfout gedetecteerd wordt op regel 16... Met in GDB *display i* vinden we

```
1: i = 60
```

6.2 Fout door vergeten te initializeren

Listing 2. 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 bovenstaand programma zit er een subtiele fout, die met GDB niet wordt herkend.

```
==2079== valgrind-1.0.2, a memory error detector for x86 GNU/Linux.
==2079== Copyright (C) 2000-2002, and GNU GPL'd, by Julian Seward.
==2079== Estimated CPU clock rate is 1800 MHz
==2079== For more details, rerun with: -v
==2079==
==2079== Use of uninitialised value of size 8
==2079==    at 0x402650BA: (within /lib/i686/libm-2.2.5.so)
==2079==    by 0x804856A: main (in /fozzie/v/frooms/develop/c/cles/val3)
==2079==    by 0x40297082: (within /lib/i686/libc-2.2.5.so)
==2079==    by 0x8048441: (within /fozzie/v/frooms/develop/c/cles/val3)
==2079==
==2079== ---- Attach to GDB ? --- [Return/N/n/Y/y/C/c] ---- y
```

Hier gaat weer iets mis: we bevestigen dat we GDB willen inschakelen, en gaan weer met *up* tot we zien welke instructie in onze code de fout veroorzaakt heeft.

```
==2079== starting GDB with cmd: /usr/bin/gdb -nw /proc/2079/exe 2079
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 2079
Reading symbols from /usr/lib/valgrind/valgrind.so...done.
Loaded symbols for /usr/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/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 /usr/lib/valgrind/libpthread.so.0...done.
Loaded symbols for /usr/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=2083, arg2=0, arg3=0)
    at vg_mylibc.c:92
92      vg_mylibc.c: No such file or directory.
        in vg_mylibc.c
(gdb) up
#1  0x00000823 in ?? ()
(gdb)
#2  0x4005633d in vgPlain_start_GDB_whilst_on_client_stack () at vg_main.c:1379
1379    vg_main.c: No such file or directory.
        in vg_main.c
(gdb)
#3  0x4005e958 in vgPlain_swizzle_esp_then_start_GDB ()
   from /usr/lib/valgrind/valgrind.so
(gdb)
#4  0x402650ba in exp () from /lib/i686/libm.so.6
(gdb)
#5  0x0804856a in main (argc=2, argv=0xbffff594) at val3.c:15
15        l = 5.0 * exp(k);
```

Hier zien we wat er aan de hand is: we proberen l te berekenen, maar die hangt af van een waarde k die niet geïnitializeerd is. Als argument wordt een waarde aan het programma meegegeven, maar enkel als deze 0.1 of 0.2 is, krijgt k een waarde. In ons geval werd k ongeïnitializeerd meegegeven om$l$ te berekenen.

6.3 Opsporen van memory leaks

Listing 3. 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 bovenstaand voorbeeldje wordt er in een functie een blok geheugen gealloceerd, maar niet meer vrijgegeven. Deze functie wordt iteratief aangeroepen, waardoor er steeds meer geheugen gereserveerd wordt zonder dat het effectief gebruikt wordt.

We compileren met

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

en gaan na wat valgrind ons te melden heeft over dit programma met

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

De uitvoer van valgrind ziet er nu als volgt uit

```
==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==
```

waaruit we kunnen afleiden dat in de functie *fun1* er geheugen wordt gealloceerd, maar niet meer vrijgegeven wordt.