Chapter 11

Debugging

> Debugging is like being the detective in a crime movie where you are also the murderer.
> (Filipe Fortes, 2013)

When a program misbehaves, *debugging* is the process of finding out *why*. There are various strategies of finding errors in a program. The crudest one is debugging by print statements. If you have a notion of where in your code the error arises, you can edit your code to insert print statements, recompile, rerun, and see if the output gives you any suggestions. There are several problems with this:

- The edit/compile/run cycle is time consuming, especially since
- often the error will be caused by an earlier section of code, requiring you to edit, compile, and rerun repeatedly. Furthermore,
- the amount of data produced by your program can be too large to display and inspect effectively, and
- if your program is parallel, you probably need to print out data from all processors, making the inspection process very tedious.

For these reasons, the best way to debug is by the use of an interactive *debugger*, a program that allows you to monitor and control the behavior of a running program. In this section you will familiarize yourself with *gdb* and *lldb*, the open source debuggers of the *GNU* and *clang* projects respectively. Other debuggers are proprietary, and typically come with a compiler suite. Another distinction is that gdb is a commandline debugger; there are graphical debuggers such as *ddd* (a frontend to gdb) or *DDT* and *TotalView* (debuggers for parallel codes). We limit ourselves to gdb, since it incorporates the basic concepts common to all debuggers.

In this tutorial you will debug a number of simple programs with gdb and valgrind. The files can be found in the repository in the directory `code/gdb`.

11.1 Compiling for debug

You often need to recompile your code before you can debug it. A first reason for this is that the binary code typically knows nothing about what variable names corresponded to what memory locations, or what lines in the source to what instructions. In order to make the binary executable know this, you have to include the *symbol table* in it, which is done by adding the `-g` option to the compiler line.

Table 11.1: List of common gdb / lldb commands.

gdb	lldb
Starting a debugger run	
`$ gdb program` `(gdb) run`	`$ lldb program` `(lldb) run`
Displaying a stack trace	
`(gdb) where`	`(lldb) thread backtrace`
Investigate a specific frame	
`frame 2`	`frame select 2`
Run/step	
`run / step / continue`	thread continue / step-in/over/out
Set a breakpoint at a line	

`break foo.c:12`
`break foo.c:12 if n>0`
`info breakpoints`
	`breakpoint set [ -f foo.c ] -l 12`

Set a breakpoint for exceptions	
`catch throw`	`break set -E C++`

Usually, you also need to lower the *compiler optimization level*: a production code will often be compiled with flags such as `-O2` or `-Xhost` that try to make the code as fast as possible, but for debugging you need to replace this by `-O0` ('oh-zero'). The reason is that higher levels will reorganize your code, making it hard to relate the execution to the source[1].

11.2 Invoking the debugger

There are three ways of using gdb: using it to start a program, attaching it to an already running program, or using it to inspect a *core dump*. We will only consider the first possibility.

Starting a debugger run	
gdb	lldb
`$ gdb program` `(gdb) run`	`$ lldb program` `(lldb) run`

Here is an example of how to start gdb with program that has no arguments (Fortran users, use `hello.F`):

tutorials/gdb/c/hello.c

```
#include <stdlib.h>
#include <stdio.h>
int main() {
  printf("hello world\n");
  return 0;
}

%% cc -g -o hello hello.c
# regular invocation:
%% ./hello
hello world
# invocation from gdb:
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... [version info]
Copyright 2004 Free Software Foundation, Inc. .... [copyright info] ....
(gdb) run
Starting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. done
hello world

Program exited normally.
(gdb) quit
%%
```

1. Typically, actual code motion is done by `-O3`, but at level `-O2` the compiler will inline functions and make other simplifications.

Important note: the program was compiled with the *debug flag* -g. This causes the *symbol table* (that is, the translation from machine address to program variables) and other debug information to be included in the binary. This will make your binary larger than strictly necessary, but it will also make it slower, for instance because the compiler will not perform certain optimizations[2].

To illustrate the presence of the symbol table do

```
%% cc -g -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

and compare it with leaving out the -g flag:

```
%% cc -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

For a program with commandline input we give the arguments to the run command (Fortran users use say.F):

tutorials/gdb/c/say.c

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv) {
  int i;
  for (i=0; i<atoi(argv[1]); i++)
    printf("hello world\n");
  return 0;
}
```

```
%% cc -o say -g say.c
%% ./say 2
hello world
hello world
%% gdb say
.... the usual messages ...
(gdb) run 2
Starting program: /home/eijkhout/tutorials/gd
Reading symbols for shared libraries +. done
hello world
hello world

Program exited normally.
```

11.3 Finding errors: where, frame, print

Let us now consider some programs with errors.

2. Compiler optimizations are not supposed to change the semantics of a program, but sometimes do. This can lead to the nightmare scenario where a program crashes or gives incorrect results, but magically works correctly with compiled with debug and run in a debugger.

11.3.1 C programs

The following code has several errors. We will use the debugger to uncover them.

```
// square.c
  int nmax,i;
  float *squares,sum;

  fscanf(stdin,"%d",nmax);
  for (i=1; i<=nmax; i++) {
    squares[i] = 1./(i*i); sum += squares[i];
  }
  printf("Sum: %e\n",sum);
```

```
%% cc -g -o square square.c
%% ./square
5000
Segmentation fault
```

The *segmentation fault* (other messages are possible too) indicates that we are accessing memory that we are not allowed to, making the program exit. A debugger will quickly tell us where this happens:

```
%% gdb square
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x0000000000000eb4a
0x00007fff824295ca in __svfscanf_l ()
```

Apparently the error occurred in a function `__svfscanf_l`, which is not one of ours, but a system function. Using the `backtrace` (or `bt`, also `where` or `w`) command we display the *call stack*. This usually allows us to find out where the error lies:

Displaying a stack trace	
gdb	lldb
`(gdb) where`	`(lldb) thread backtrace`

```
(gdb) where
#0  0x00007fff824295ca in __svfscanf_l ()
#1  0x00007fff8244011b in fscanf ()
#2  0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at square.c:7
```

We inspect the actual problem:

Investigate a specific frame	
gdb	clang
`frame 2`	`frame select 2`

We take a close look at line 7, and see that we need to change `nmax` to `&nmax`.

There is still an error in our program:

```
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x000000010000f000
0x0000000100000ebe in main (argc=2, argv=0x7fff5fbfc7a8) at square1.c:9
9           squares[i] = 1./(i*i); sum += squares[i];
```

We investigate further:

```
(gdb) print i
$1 = 11237
(gdb) print squares[i]
Cannot access memory at address 0x10000f000
(gdb) print squares
$2 = (float *) 0x0
```

and we quickly see that we forgot to allocate `squares`.

By the way, we were lucky here: this sort of memory errors is not always detected. Starting our programm with a smaller input does not lead to an error:

```
(gdb) run
50
Sum: 1.625133e+00

Program exited normally.
```

Memory errors can also occur if we have a legitimate array, but we access it outside its bounds. The following program fills an array, forward, and reads it out, backward. However, there is an indexing error in the second loop.

```
// up.c
  int nlocal = 100,i;
  double s, *array = (double*) malloc(nlocal*sizeof(double));
  for (i=0; i<nlocal; i++) {
    double di = (double)i;
    array[i] = 1/(di*di);
  }
  s = 0.;
  for (i=nlocal-1; i>=0; i++) {
    double di = (double)i;
    s += array[i];
  }
```

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
```

```
Reason: KERN_INVALID_ADDRESS at address: 0x0000000100200000
0x0000000100000f43 in main (argc=1, argv=0x7fff5fbfe2c0) at up.c:15
15          s += array[i];
(gdb) print array
$1 = (double *) 0x100104d00
(gdb) print i
$2 = 128608
```

You see that the index where the debugger finally complains is quite a bit larger than the size of the array.

Exercise 11.1. Can you think of a reason why indexing out of bounds is not immediately fatal? What would determine where it does become a problem? (Hint: how is computer memory structured?)

In section 11.8 you will see a tool that spots any out-of-bound indexing.

11.3.2 Fortran programs

Compile and run the following program:

MISSING SNIPPET gdb-squaref

It should end prematurely with a message such as 'Illegal instruction'. Running the program in gdb quickly tells you where the problem lies:

```
(gdb) run
Starting program: tutorials/gdb//fsquare
Reading symbols for shared libraries ++++. done

Program received signal EXC_BAD_INSTRUCTION,
Illegal instruction/operand.
0x0000000100000da3 in square () at square.F:7
7                 sum = sum + squares(i)
```

We take a close look at the code and see that we did not allocate `squares` properly.

11.4 Stepping through a program

Stepping through a program

gdb	lldb	meaning
`run`		start a run
`cont`		continue from breakpoint
`next`		next statement on same level
`step`		next statement, this level or next

Often the error in a program is sufficiently obscure that you need to investigate the program run in detail. Compile the following program

```
// roots.c
float root(int n)
{
  float r;
  r = sqrt(n);
  return r;
}

int main() {
  feenableexcept(FE_INVALID | FE_OVERFLOW);
  int i;
  float x=0;
  for (i=100; i>-100; i--)
    x += root(i+5);
  printf("sum: %e\n",x);
```

and run it:

```
%% ./roots
sum: nan
```

Start it in gdb as before:

```
%% gdb roots
GNU gdb 6.3.50-20050815
Copyright 2004 Free Software Foundation, Inc.
....
```

but before you run the program, you set a *breakpoint* at `main`. This tells the execution to stop, or 'break', in the main program.

```
(gdb) break main
Breakpoint 1 at 0x100000ea6: file root.c, line 14.
```

Now the program will stop at the first executable statement in `main`:

```
(gdb) run
Starting program: tutorials/gdb/c/roots
Reading symbols for shared libraries +. done

Breakpoint 1, main () at roots.c:14
14        float x=0;
```

Most of the time you will set a breakpoint at a specific line:

Set a breakpoint at a line	
gdb	lldb
`break foo.c:12`	`breakpoint set [ -f foo.c ] -l 12`

If execution is stopped at a breakpoint, you can do various things, such as issuing the `step` command:

```
Breakpoint 1, main () at roots.c:14
14        float x=0;
(gdb) step
15        for (i=100; i>-100; i--)
(gdb)
16          x += root(i);
(gdb)
```

(if you just hit return, the previously issued command is repeated). Do a number of `steps` in a row by hitting return. What do you notice about the function and the loop?

Switch from doing `step` to doing `next`. Now what do you notice about the loop and the function?

Set another breakpoint: `break 17` and do `cont`. What happens?

Rerun the program after you set a breakpoint on the line with the `sqrt` call. When the execution stops there do `where` and `list`.

- If you set many breakpoints, you can find out what they are with `info breakpoints`.
- You can remove breakpoints with `delete n` where `n` is the number of the breakpoint.
- If you restart your program with `run` without leaving gdb, the breakpoints stay in effect.
- If you leave gdb, the breakpoints are cleared but you can save them: `save breakpoints <file>`. Use `source <file>` to read them in on the next gdb run.

11.5 Inspecting values

Run the previous program again in gdb: set a breakpoint at the line that does the `sqrt` call before you actually call `run`. When the program gets to line 8 you can do `print n`. Do `cont`. Where does the program stop?

If you want to repair a variable, you can do `set var=value`. Change the variable `n` and confirm that the square root of the new value is computed. Which commands do you do?

11.6 Breakpoints

If a problem occurs in a loop, it can be tedious keep typing `cont` and inspecting the variable with `print`. Instead you can add a condition to an existing breakpoint. First of all, you can make the breakpoint subject to a condition: with

```
condition 1 if (n<0)
```

breakpoint 1 will only obeyed if `n<0` is true.

You can also have a breakpoint that is only activated by some condition. The statement

```
break 8 if (n<0)
```

means that breakpoint 8 becomes (unconditionally) active after the condition `n<0` is encountered.

Set a breakpoint	
gdb	lldb
`break foo.c:12`	`breakpoint set [ -f foo.c ] -l 12`
`break foo.c:12 if n>0`	

Remark 9 *You can break on* `NaN` *with the following trick:*

```
break foo.c:12 if x!=x
```

using the fact that `NaN` *is the only number not equal to itself.*

Another possibility is to use `ignore 1 50`, which will not stop at breakpoint 1 the next 50 times.

Remove the existing breakpoint, redefine it with the condition `n<0` and rerun your program. When the program breaks, find for what value of the loop variable it happened. What is the sequence of commands you use?

You can set a breakpoint in various ways:

- `break foo.c` to stop when code in a certain file is reached;
- `break 123` to stop at a certain line in the current file;
- `break foo` to stop at subprogram `foo`
- or various combinations, such as `break foo.c:123`.

Information about breakpoints:

- If you set many breakpoints, you can find out what they are with `info breakpoints`.
- You can remove breakpoints with `delete n` where `n` is the number of the breakpoint.
- If you restart your program with `run` without leaving gdb, the breakpoints stay in effect.
- If you leave gdb, the breakpoints are cleared but you can save them: `save breakpoints <file>`. Use `source <file>` to read them in on the next gdb run.
- In languages with *exceptions*, such as *C++*, you can set a *catchpoint*:

Set a breakpoint for exceptions	
gdb	clang
`catch throw`	`break set -E C++`

Finally, you can execute commands at a breakpoint:

```
break 45
command
print x
cont
end
```

This states that at line 45 variable `x` is to be printed, and execution should immediately continue.

If you want to run repeated gdb sessions on the same program, you may want to save an reload breakpoints. This can be done with

```
save-breakpoint filename
source filename
```

11.7 Memory debugging

Many problems in programming stem from memory errors. We start with a sort description of the most common types, and then discuss tools that help you detect them.

11.7.1 Type of memory errors

11.7.1.1 Invalid pointers

Dereferencing a pointer that does not point to an allocated object can lead to an error. If your pointer points into valid memory anyway, your computation will continue but with incorrect results.

However, it is more likely that your program will probably exit with a *segmentation violation* or a *bus error*.

11.7.1.2 Out-of-bounds errors

Addressing outside the bounds of an allocated object is less likely to crash your program and more likely to give incorrect results.

Exceeding bounds by a large enough amount will again give a segmentation violation, but going out of bounds by a small amount may read invalid data, or corrupt data of other variables, giving incorrect results that may go undetected for a long time.

11.7.1.3 Memory leaks

We speak of a *memory leak* if allocated memory becomes unreachable. Example:

```
if (something) {
  double *x = malloc(10*sizeofdouble);
  // do something with x
}
```

After the conditional, the allocated memory is not freed, but the pointer that pointed to has gone away.

This last type especially can be hard to find. Memory leaks will only surface in that your program runs out of memory. That in turn is detectable because your allocation will fail. It is a good idea to always check the return result of your *malloc* or *allocate* statement!

11.8 Memory debugging with Valgrind

Insert the following allocation of squares in your program:

```
squares = (float *) malloc( nmax*sizeof(float) );
```

Compile and run your program. The output will likely be correct, although the program is not. Can you see the problem?

To find such subtle memory errors you need a different tool: a memory debugging tool. A popular (because open source) one is *valgrind*; a common commercial tool is *purify*.

```c
// square1.c
int main(int argc,char **argv) {
  int nmax,i;
  float *squares,sum;

  fscanf(stdin,"%d",&nmax);
  squares = (float*) malloc(nmax*sizeof(float));
  for (i=1; i<=nmax; i++) {
    squares[i] = 1./(i*i);
    sum += squares[i];
  }
  printf("Sum: %e\n",sum);
```

Compile this program with `cc -o square1 square1.c` and run it with `valgrind square1` (you need to type the input value). You will lots of output, starting with:

```
%% valgrind square1
==53695== Memcheck, a memory error detector
==53695== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==53695== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==53695== Command: a.out
==53695==
10
==53695== Invalid write of size 4
==53695==    at 0x100000EB0: main (square1.c:10)
==53695==  Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695==    at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695==    by 0x100000E77: main (square1.c:8)
==53695==
==53695== Invalid read of size 4
==53695==    at 0x100000EC1: main (square1.c:11)
==53695==  Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
```

```
==53695==    at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695==    by 0x100000E77: main (square1.c:8)
```

Valgrind is informative but cryptic, since it works on the bare memory, not on variables. Thus, these error messages take some exegesis. They state that a line 10 writes a 4-byte object immediately after a block of 40 bytes that was allocated. In other words: the code is writing outside the bounds of an allocated array. Do you see what the problem in the code is?

Note that valgrind also reports at the end of the program run how much memory is still in use, meaning not properly `free`d.

If you fix the array bounds and recompile and rerun the program, valgrind still complains:

```
==53785== Conditional jump or move depends on uninitialised value(s)
==53785==    at 0x10006FC68: __dtoa (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x10003199F: __vfprintf (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x1000738AA: vfprintf_l (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x1000A1006: printf (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x100000EF3: main (in ./square2)
```

Although no line number is given, the mention of `printf` gives an indication where the problem lies. The reference to an 'uninitialized value' is again cryptic: the only value being output is `sum`, and that is not uninitialized: it has been added to several times. Do you see why valgrind calls it uninitialized all the same?

11.8.1 Electric fence

The *electric fence* library is one of a number of tools that supplies a new *`malloc`* with debugging support. These are linked instead of the `malloc` of the standard *`libc`*.

```
cc -o program program.c -L/location/of/efence -lefence
```

Suppose your program has an out-of-bounds error. Running with gdb, this error may only become apparent if the bounds are exceeded by a large amount. On the other hand, if the code is linked with *`libefence`*, the debugger will stop at the very first time the bounds are exceeded.

11.9 Further reading

A good tutorial: http://www.dirac.org/linux/gdb/.

Reference manual: http://www.ofb.net/gnu/gdb/gdb_toc.html.