The First Real Bug

0800  Started

1000  Stopped - asked

13:00  HP - MC 1.213264915 4.615928576 (1)

033  PRO 2  2.120427645

Relays 6-2 in 033 found special speed test
in Federal. Relay changed.

1130  Started Cosine Tape (Sine check)

1525  Started Mutley Adder Test

1545  Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

1600  Antenna started

1700  Closed down.
Debugging vs Testing

*Software testing* is any activity aimed at evaluating an attribute or capability of a program and determining whether it meets its specified results

All about "does it work"?

*Debugging* is a methodical process of finding and reducing the number of bugs, or defects, in a computer program …, thus making it behave as expected

All about "why does it not work" and "what can we do about that"?

They are fundamentally different activities.

Testing can indicate the need to debug, but often provides only superficial clues as to the location or nature of the error.
printf() as an Aid

Perhaps the simplest approach to debugging is to add output code to the program in order to display the values of selected variables and indicate flow of control as the program executes.

This is often referred to as *instrumenting* the code.

- Easy to apply.
- Use preprocessor directives to enable/disable diagnostic output.
- Lets the code tell you what is actually happening, as opposed to what you believe is happening – psychological issues often hinder debugging.
- Can be cumbersome and difficult to "tune".

This technique is often undervalued and often overvalued.
gdb is a system tool that allows the user to:

- Step through the execution of a program, instruction by instruction.
- View and even modify the values of variables.
- Set *breakpoints* that cause the execution of a program to be halted at specific places in the code.
- Set *watchpoints* that cause the execution of a program to be halted whenever the value of a user-defined expression changes.
- Show a list of the active stack frames.
- Display a range of source code lines.
- Disassemble the current machine code to assembly language.

… and more.
Some gdb Resources

*The Art of Debugging with GDB, DDD, and Eclipse*,
N Matloff & P J Salzman,
No Starch Press (c)2008
ISBN 978-1-593-27174-9

Some reasonably good gdb cheatsheets:

http://www.yolinux.com/TUTORIALS/GDB-Commands.html
The C source for our running example follows… it is adapted from an example by Norman Matloff (http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html):

```c
#include <stdio.h>
#include <stdbool.h>

/* prime-number finding program    
   Will (after bugs are fixed) report a list of all primes which are less than or equal to the user-supplied upper bound. 
   This code is riddled with errors! */

#define MAXPRIMES 100
bool Prime[MAXPRIMES];  /* Prime[I] will be true if I is prime, false otherwise */

void CheckPrime(int K);
```
Example Program

```c
... int main() {

    int N;
    int UpperBound; /* we will check all numbers up through this one for primeness */

    printf("enter upper bound\n");
    scanf("%d", UpperBound);

    Prime[2] = true;

    for ( N = 3; N <= UpperBound; N += 2 )
        CheckPrime(N);
        if ( Prime[N] ) printf("%d is a prime\n",N);

    return 0;
}
```
void CheckPrime(int K) {

    int J;

    /* the plan: see if J divides K, for all values J which are

    (a) themselves prime (no need to try J if it is nonprime), and
    (b) less than or equal to sqrt(K) (if K has a divisor larger than this square root, it must also have a smaller one, so no need to check for larger ones)
*/
Example Program

```c
... 

J = 2;
while ( true ) {
    if ( Prime[J] )
        if ( K % J == 0 ) {
            Prime[K] = false;
            return;
        }
    J++;
}

/* if we get here, then there were no divisors of K, so 
K must be prime */

Prime[K] = true;
}
```
Compiling for Debugging

In order to take full advantage of gdb's features, you should generally:

- disable code optimizations by using –O0.
- enable the generation of extra debugging information by using –g, or better, by using –ggdb3.

So, in this case, I compiled the preceding source code using the command line:

```
gcc -o matloff1 -std=c99 -O0 -ggdb3 -m32 matloff1.c
```

This results in one compiler warning, which I unwisely ignore…
I executed the program by typing the command `matloff1`.

The program prompts the user for a bound on the number of values to be checked; I entered the value 20.

The continuing execution of the program resulted in the following message:

```
Segmentation fault
```

This indicates a runtime error related to an impermissible access to memory… but why?
Starting gdb

Start the debugger by typing the command `gdb matloff1`.

gdb starts up with a copyright message and then displays a user prompt:

```
$ gdb matloff1
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 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.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>
Reading symbols from /home/williammcquain/2505/gdb/matloff1/matloff1...done.
(gdb)
```
Running the Program

Begin execution of the program by entering the `run` command, then respond to the user prompt:

```
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/williammcquain/2505/gdb/matloffl/matloffl...done.
(gdb) run
Starting program: /home/williammcquain/2505/gdb/matloffl/matloffl
enter upper bound
20

Program received signal SIGSEGV, Segmentation fault.
0xf7eab0cb in _IO_vfscanf () from /lib32/libc.so.6
(gdb)
```

Now, this gives us some information, including the address of the (machine) instruction that caused the error, and the function in which the error occurred.

But `_IO_vfscanf()` is a system function, not user code…
We can get more information about how we arrived at the error by using backtrace:

```
(gdb) run
Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1
enter upper bound
20

Program received signal SIGSEGV, Segmentation fault.
0xff7eab0cb in _IO_vfscanf () from /lib32/libc.so.6
(gdb) backtrace
#0 0xff7eab0cb in _IO_vfscanf () from /lib32/libc.so.6
#1 0xff7eb0b9e in __isoc99_scanf () from /lib32/libc.so.6
#2 0x0804846e in main () at matloff1.c:23
(gdb) 
```

This shows the stack contains three stack frames at the time the error occurs, and provides the crucial information that:

```
line 23 in main() called __isoc99_scanf(),
which called __IO_vfscanf()
```

It seems unlikely either of the latter functions is incorrect… what's line 23?
We can display the relevant source by using `list`:

```
(int) list matlof1.c:23
18    int N;
19    int UpperBound;    /* we will check all number up through this one
20                          primeness */
21
22    printf("enter upper bound\n");
23    scanf("%d", UpperBound);
24
26
---Type <return> to continue, or q <return> to quit---
```

In this case, the error should be obvious, we passed the value of `UpperBound` to `scanf()` instead of passing the address of `UpperBound`…

… and `scanf()` then treated that value as an address… with unpleasant results.
Before modifying the source code and rebuilding, we need to stop the running process, by using the `kill` command:

```
20
21
22
23
24
25
26

---Type <return> to continue, or q <return> to quit---
Quit
(gdb) kill
Kill the program being debugged? (y or n) y
```
Fix the First Bug

We fix the error by inserting the address-of operator:

```c
int main() {
    scanf("%d", &UpperBound);
}
```

Now, rebuild as before and try running the program again…

Segmentation fault

Note: I opened a second terminal window to perform the rebuild and test the program again… that saves the time to exit and restart gdb (of course, in this case I knew in advance there were more bugs).
Running the Program Again

Restart the program within gdb and see what happens:

This time we got better information because the source for matloff1.c is available.

We know:
- CheckPrime() was called with \( K = 3 \)
- The error occurred in evaluating \( \text{Prime}[j] \)
As before, let's see what the surrounding code is:

```
J = 2;
while ( true ) {
    if ( Prime[J] )
        if ( K % J == 0 ) {
            Prime[K] = false;
            return;
        }
}
```

Hm… that's somewhat informative. Apparently $J$ must be out of bounds.
We can see the value of a variable by using the command `print`:

```
42                so no need to check for larger ones) */
43
44                J = 2;
45                while ( true ) {
46                      if ( Prime[J] )
47                          if ( K % J == 0 ) {
48                              Prime[K] = false;
49                              return;
50                      }
50 (gdb) print J
51 $1 = 4032
50 (gdb)
```

Well, `Prime[]` is of dimension 100, so that is certainly out of bounds… how did this happen?

Better take a somewhat wider look at the source… certainly "while (true)" looks a bit odd.
In this case, I find it easier to just switch to my text editor and see what's going on:

```c
/* the plan: see if J divides K, for all values J which are

(a) themselves prime (no need to try J if it is nonprime), and
(b) less than or equal to sqrt(K) (if K has a divisor larger than this square root, it must also have a smaller one, so no need to check for larger ones)
*/
J = 2;
while ( true ) {
    if ( Prime[J] )
        if ( K % J == 0 ) {
            Prime[K] = false;
            return;
        }
    J++;
}
```

The loop bears no resemblance to the stated plan… the code never tries to limit J to be less than or equal to \sqrt(K).
The Problem

The loop never exits unless we have a value for \( J \) such that both:
- \( \text{Prime}[J] == \text{true} \)
- \( J \) divides \( K \)

But if \( K == 3 \) then the first prime that divides \( K \) would be 3 itself.

But we know that \( J \) reached the value 4032.

Why didn't the loop exit when we reached \( J == 3 \)?

It must have been that \( \text{Prime}[3] \) was not true.

Examining the earlier source code, we see that \( \text{Prime}[3] \) will not have been explicitly set at this point.

We could fix this by assuming each \( K \) is prime until shown otherwise, and so setting \( \text{Prime}[K] \) to true before entering the function…
Fixing the Second Bug

But it's more efficient to make the loop exit once we've examined all the necessary candidates for divisors of $K$:

```c
/* the plan: see if $J$ divides $K$, for all values $J$ which
are

(a) themselves prime (no need to try $J$ if it is
nonprime), and
(b) less than or equal to $\sqrt{K}$ (if $K$ has a divisor
larger than this square root, it must also have a
smaller one, so no need to check for larger ones)
*/
for ( $J = 2$; $J * J <= K$; $J++$ ) {
    if ( Prime[$J$] )
        if ( $K \% J == 0$ ) {
            Prime[$K$] = false;
            return;
        }
    $J++$;
}
```

...
Well, no segmentation fault… but this didn't report any primes up to 20…

What to do when we have no immediate indication of what's wrong?

It would seem useful to trace the execution of the program.
gdb allows us to set *breakpoints*, that is positions at which execution will automatically halt:

```
enter upper bound
20

Program exited normally.
(gdb) break main
Breakpoint 1 at 0x804844d: file matloff1.c, line 22.
(gdb) run
Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1

Breakpoint 1, main () at matloff1.c:22
22   printf("enter upper bound\n");
```

**Important:** the displayed line of code has NOT been executed yet!
gdb also allows us to step through the program one instruction at a time:

Since line 23 is a `scanf()` call, we must enter the input value and hit return before gdb resumes by displaying the next instruction.
The **gdb** command **display** is like **print** except that the value of the specified variable is shown after each step is taken:

The initial display of $N$ makes sense (why?), as does the next.

But execution goes from line 27 to line 28 and back to line 27… that's not what we expected… (see the source for **main**()).
Fixing the Third Bug

Ah… missing braces around the intended body of the for loop:

```c
int main() {
    for (N = 3; N <= UpperBound; N += 2) {
        CheckPrime(N);
        if ( Prime[N] ) printf("%d is a prime\n",N);
    }
}
```

BTW, this is why I suggest you ALWAYS put braces around the body of a selection or loop structure.
You might want to use the `clear` command to reset the breakpoint.

OK, this looks better, but we missed the prime 2 and reported that 9 and 15 are prime.

See the source code for the reason for these final bugs…