

9/9

0800 Antan started

1000 " stopped - antan ✓

13<sup>00</sup> (033) MP-MC ~~1.982647000~~ ~~2.130476415~~ (03) 4.615925059 (-2)

(033) PRO 2 2.130476415

convct 2.130676415

Relays 6-2 in 033 failed special speed test  
in relay .. 10,000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Multi-Adder Test.

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

First actual case of bug being found.

~~1630~~ Antan started.

1700 closed down.

Relay 3375  
Relay 3376

*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.

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.

*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://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>

<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>):

```
#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

void CheckPrime(int K, bool Prime[]);
. . .
```

```
. . .
int main() {

    int N;
    int UpperBound;          /* we will check all numbers up
                               through this one for primeness */
    bool Prime[MAXPRIMES] = {0};
                               /* Prime[I] will be true if I is
                               prime, false otherwise */

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

    Prime[2] = true;

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

    return 0;
}
```

```
. . .  
void CheckPrime(int K, bool Prime[]) {  
  
    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)  
  
    */
```



```
. . .

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;
}
```

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 matloff1.c
```

This results in two compiler warnings, 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.

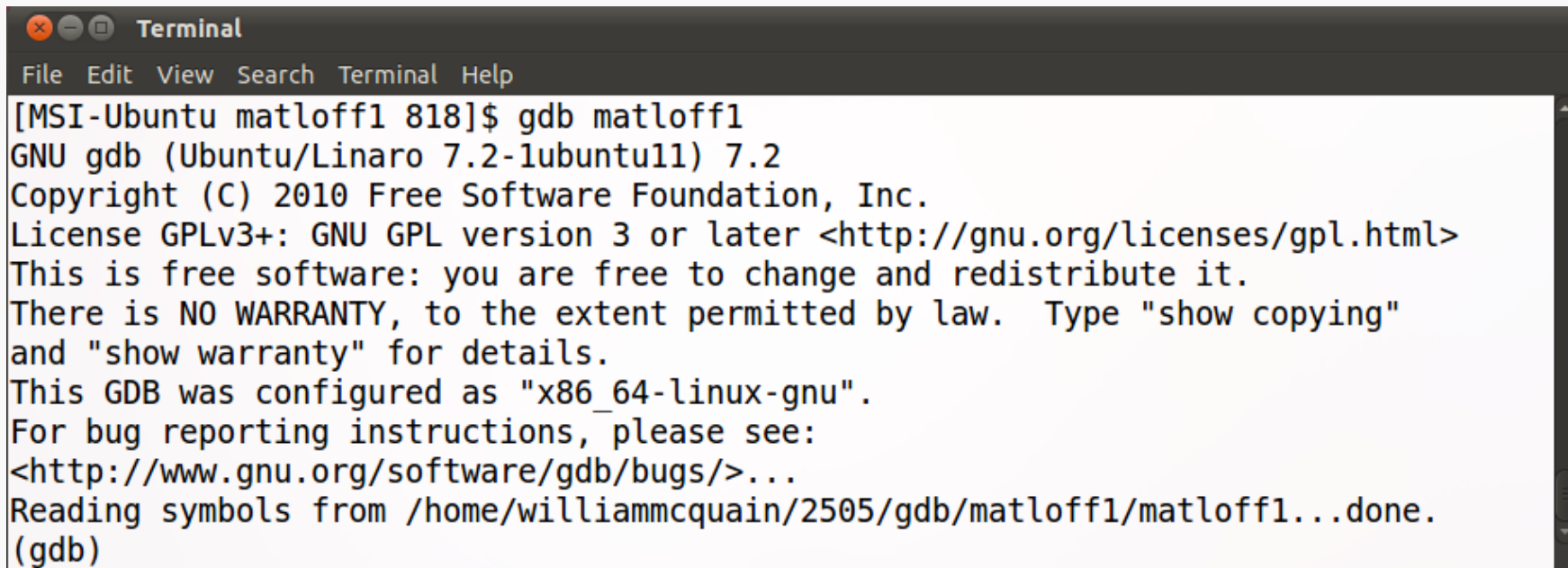
The continuing execution of the program resulted in the following message:

```
Segmentation fault
```

This indicates a runtime error related to an impermissible access to memory... but why?

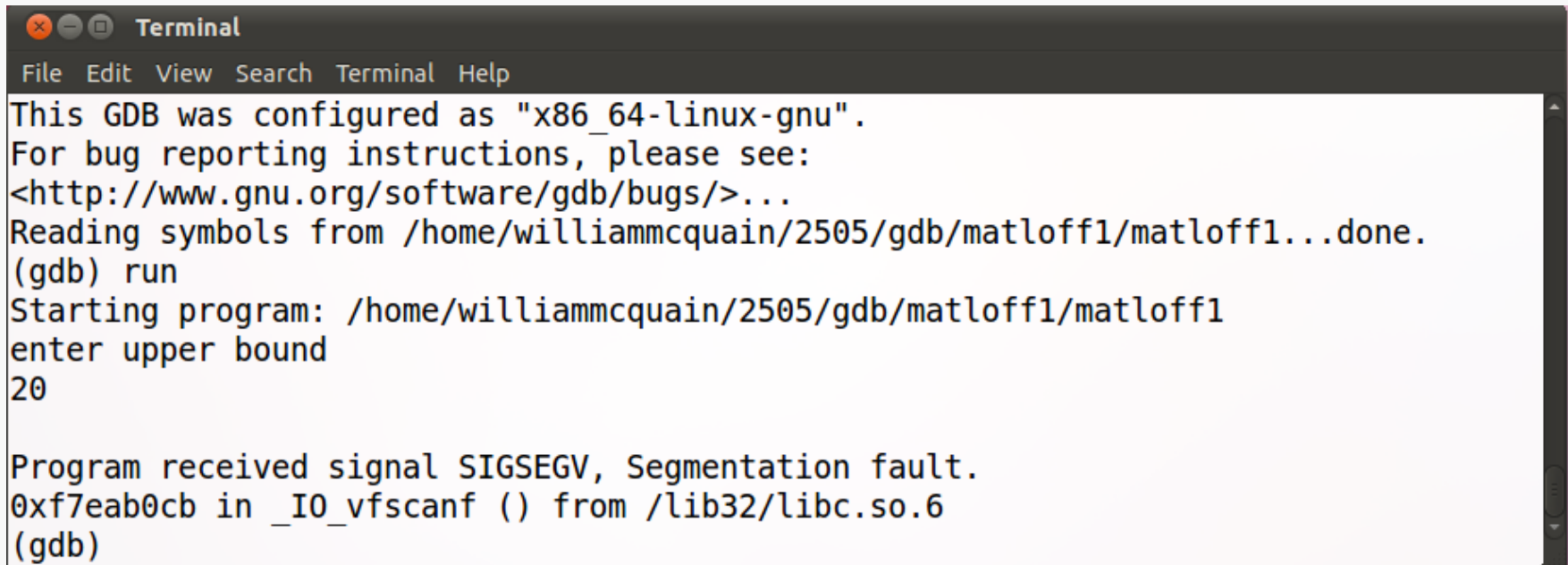
Start the debugger by typing the command `gdb matloff1`.

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

A terminal window titled "Terminal" with a menu bar containing "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal content shows the execution of the command "gdb matloff1" in a shell. The output includes the GNU gdb version (7.2), copyright information (© 2010 Free Software Foundation, Inc.), the GPL license text, and the path to the symbols for the program "matloff1". The prompt "(gdb)" is shown at the end.

```
[MSI-Ubuntu matloff1 818]$ 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)
```

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

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal output shows GDB configuration information, the user entering the 'run' command, the program starting, and the user entering '20' as the upper bound. The program then crashes with a segmentation fault (SIGSEGV) in the function `_IO_vfscanf` from `/lib32/libc.so.6`.

```
Terminal
File Edit View Search Terminal Help
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) run
Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1
enter upper bound
20

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

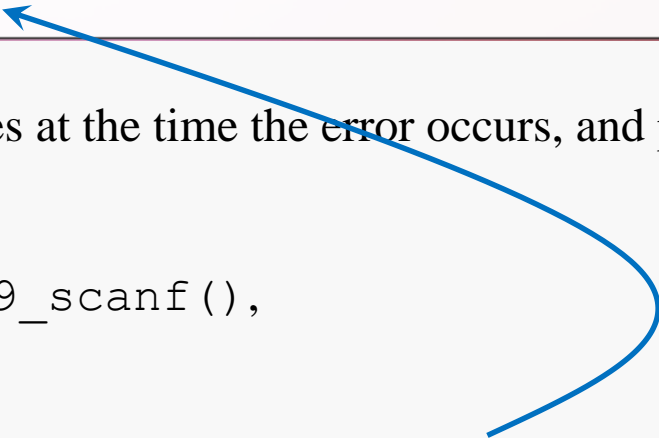
Now, this gives us some information, including the address of the (machine) instruction that caused the error, and the function in which the error occurred.

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`:

```
Terminal
File Edit View Search Terminal Help
(gdb) run
Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1
enter upper bound
20

Program received signal SIGSEGV, Segmentation fault.
0xf7eab0cb in _IO_vfscanf () from /lib32/libc.so.6
(gdb) backtrace
#0  0xf7eab0cb in _IO_vfscanf () from /lib32/libc.so.6
#1  0xf7eb0b9e in __isoc99_scanf () from /lib32/libc.so.6
#2  0x0804846e in main () at matloff1.c:23
(gdb) █
```

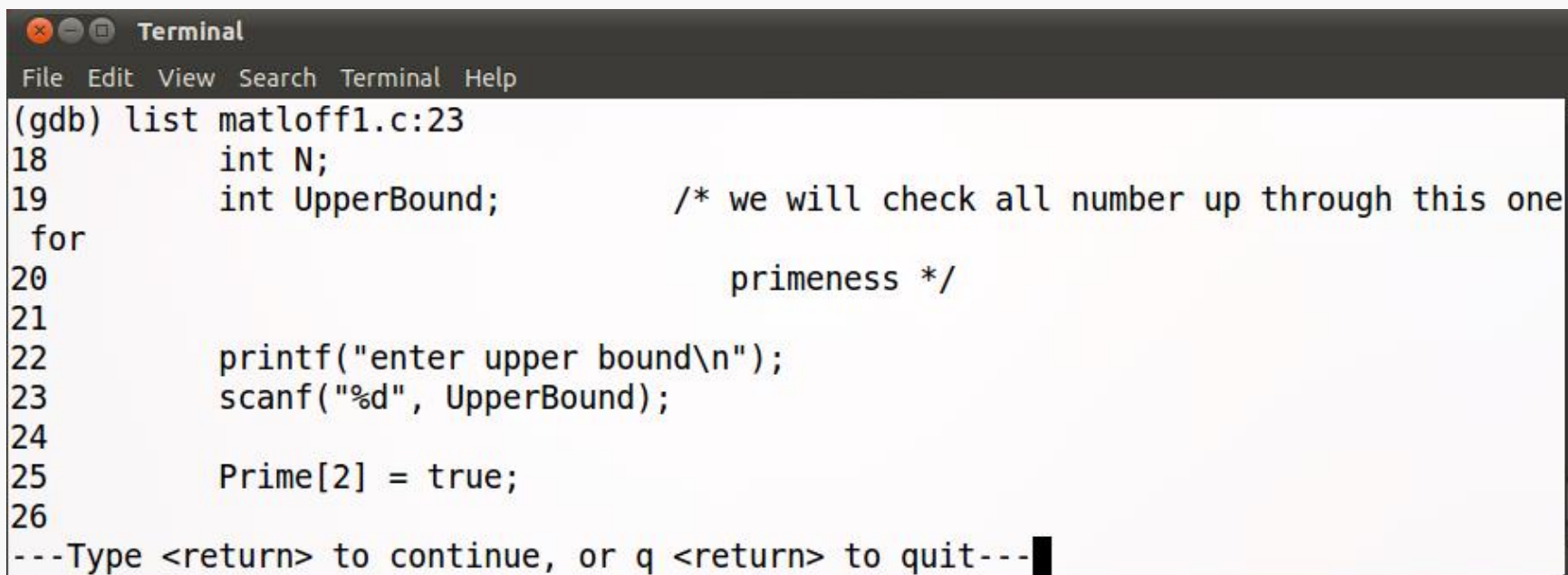


This shows the stack contains three stack frames at the time the error occurs, and provides the crucial information that:

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`:

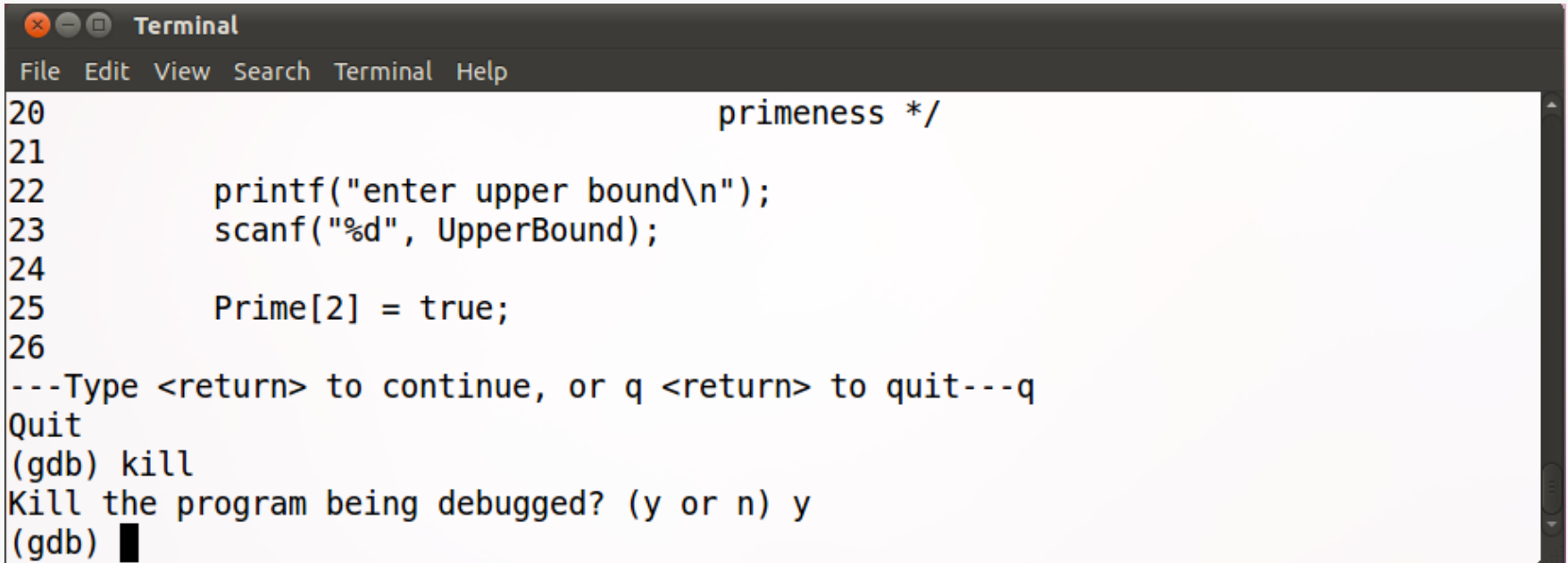
A terminal window titled "Terminal" with a menu bar containing "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows the command "(gdb) list matloff1.c:23" and the following source code:

```
18     int N;
19     int UpperBound;          /* we will check all number up through this one
   for
20                               primeness */
21
22     printf("enter upper bound\n");
23     scanf("%d", UpperBound);
24
25     Prime[2] = true;
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:

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The window contains C code for a primality test. The code is as follows:

```
20             primeness */
21
22     printf("enter upper bound\n");
23     scanf("%d", UpperBound);
24
25     Prime[2] = true;
26
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) █
```



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

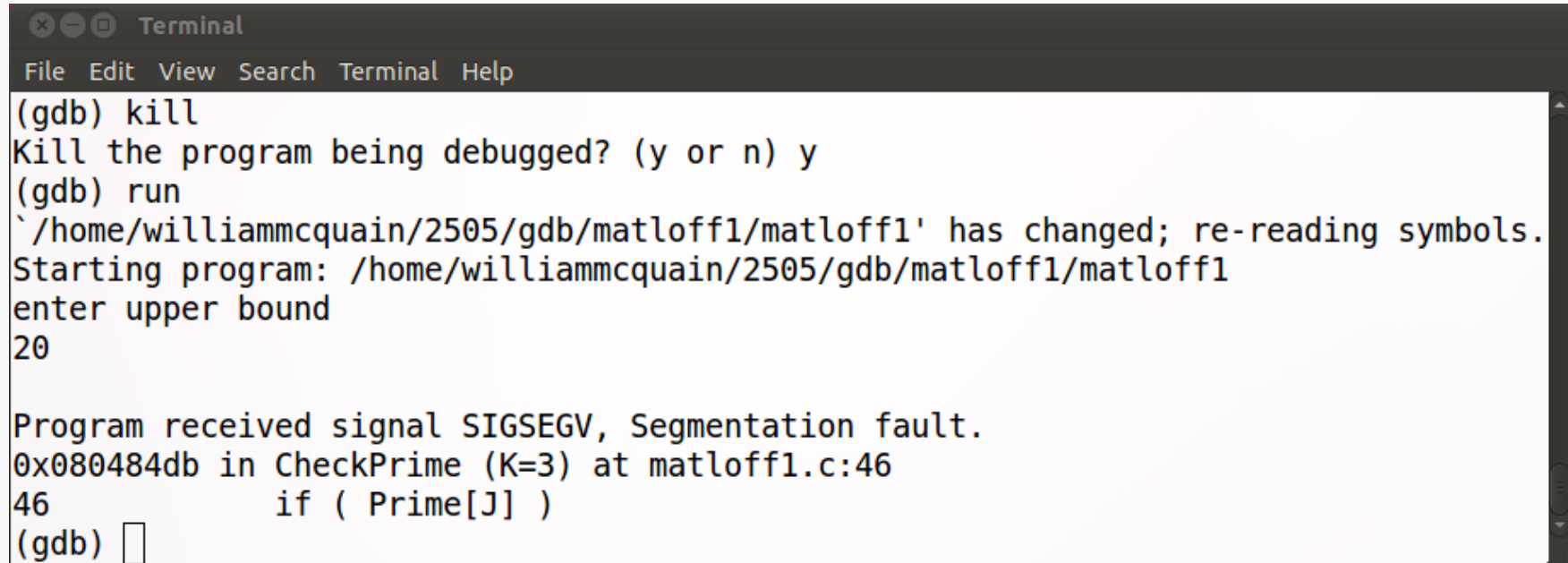
```
. . .  
int main() {  
. . .  
    scanf("%d", &UpperBound);  
. . .
```

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).

Restart the program within gdb and see what happens:



```
Terminal
File Edit View Search Terminal Help
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) run
`/home/williammcquain/2505/gdb/matloff1/matloff1' has changed; re-reading symbols.
Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1
enter upper bound
20

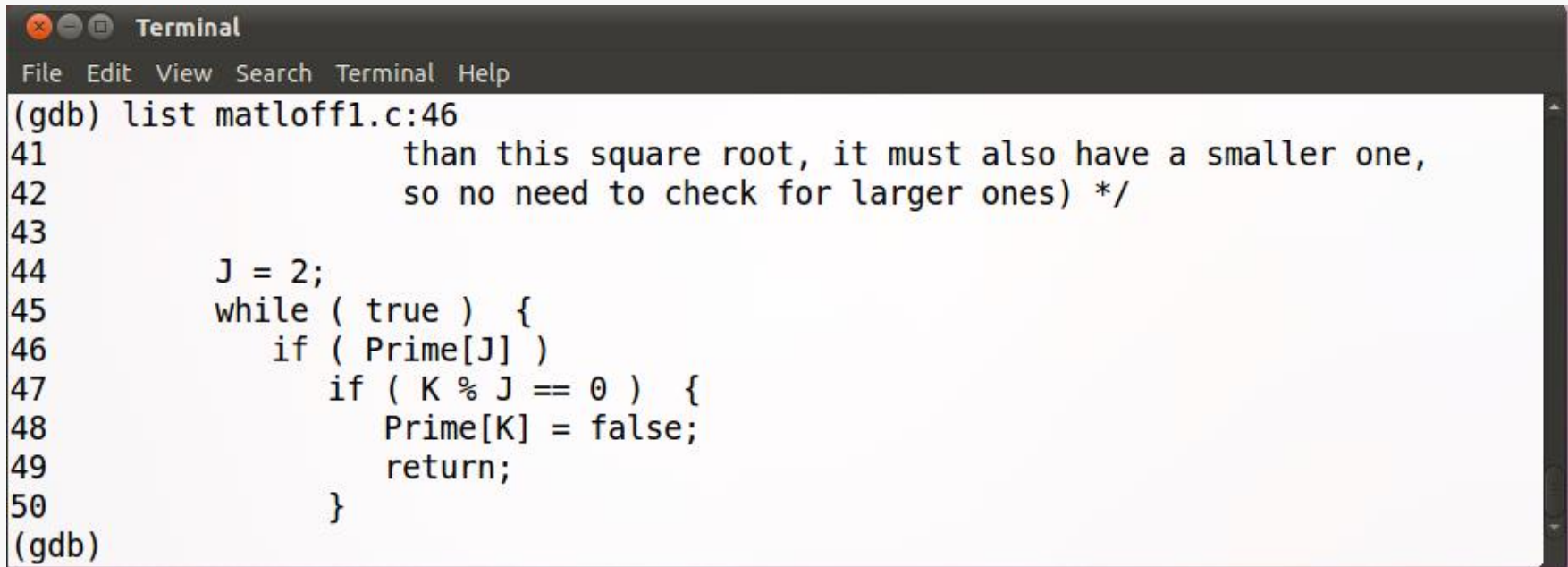
Program received signal SIGSEGV, Segmentation fault.
0x080484db in CheckPrime (K=3) at matloff1.c:46
46         if ( Prime[J] )
(gdb) █
```

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 `Prime[j]`

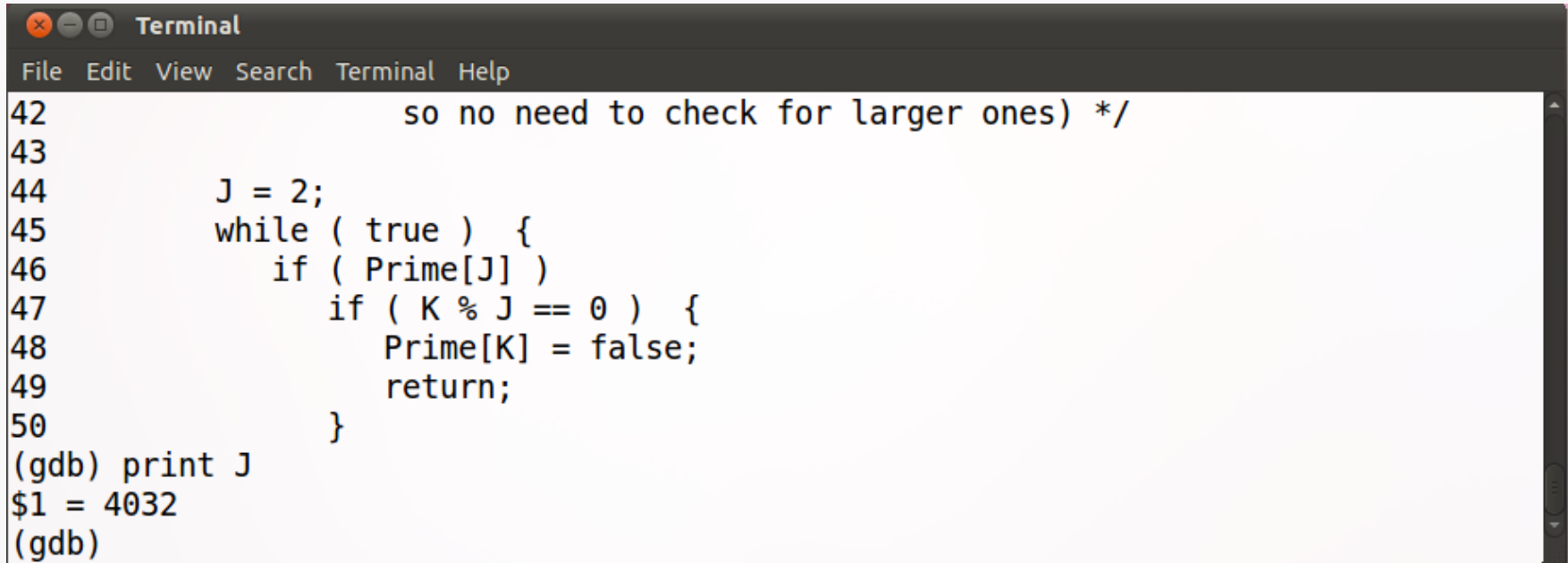
As before, let's see what the surrounding code is:



```
Terminal
File Edit View Search Terminal Help
(gdb) list matloff1.c:46
41         than this square root, it must also have a smaller one,
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             }
(gdb)
```

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`:

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The window contains C code and a gdb session. The code is a snippet of a sieve algorithm. The gdb session shows the command `print J` being executed, resulting in the output `$1 = 4032`.

```
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             }
(gdb) print J
$1 = 4032
(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:

```
. . .
/* 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  $\text{sqrt}(K)$ .

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

The loop never exits unless we have a value for J such that both:

- Prime[J] == 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 Prime[3] was not true.

Examining the earlier source code, we see that Prime[3] will not have been explicitly set at this point.

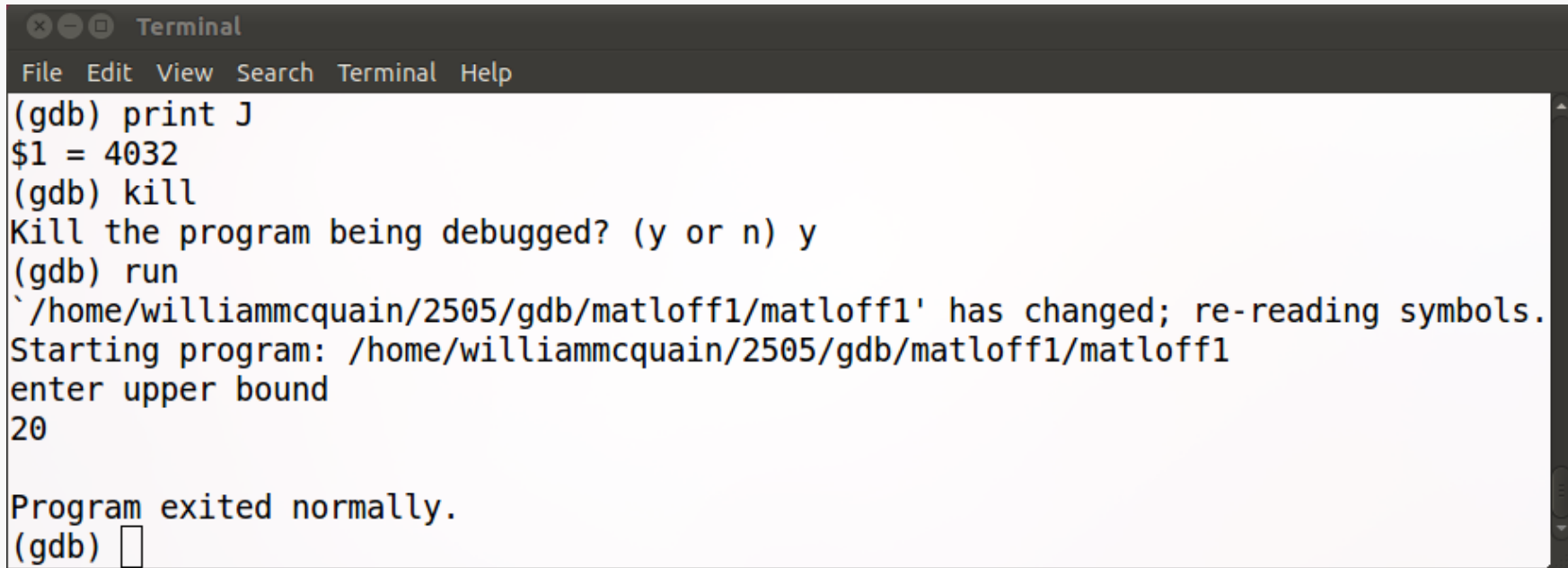
We could fix this by assuming each K is prime until shown otherwise, and so setting Prime[K] to true before entering the function...

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

```
. . .
/* 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++;
}
. . .
```

A terminal window titled "Terminal" with a menu bar containing "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal content shows a GDB session: "(gdb) print J", "\$1 = 4032", "(gdb) kill", "Kill the program being debugged? (y or n) y", "(gdb) run", "'/home/williammcquain/2505/gdb/matloff1/matloff1' has changed; re-reading symbols.", "Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1", "enter upper bound", "20", "Program exited normally.", and "(gdb) ".

```
Terminal
File Edit View Search Terminal Help
(gdb) print J
$1 = 4032
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) run
'/home/williammcquain/2505/gdb/matloff1/matloff1' has changed; re-reading symbols.
Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1
enter upper bound
20

Program exited normally.
(gdb) 
```

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:

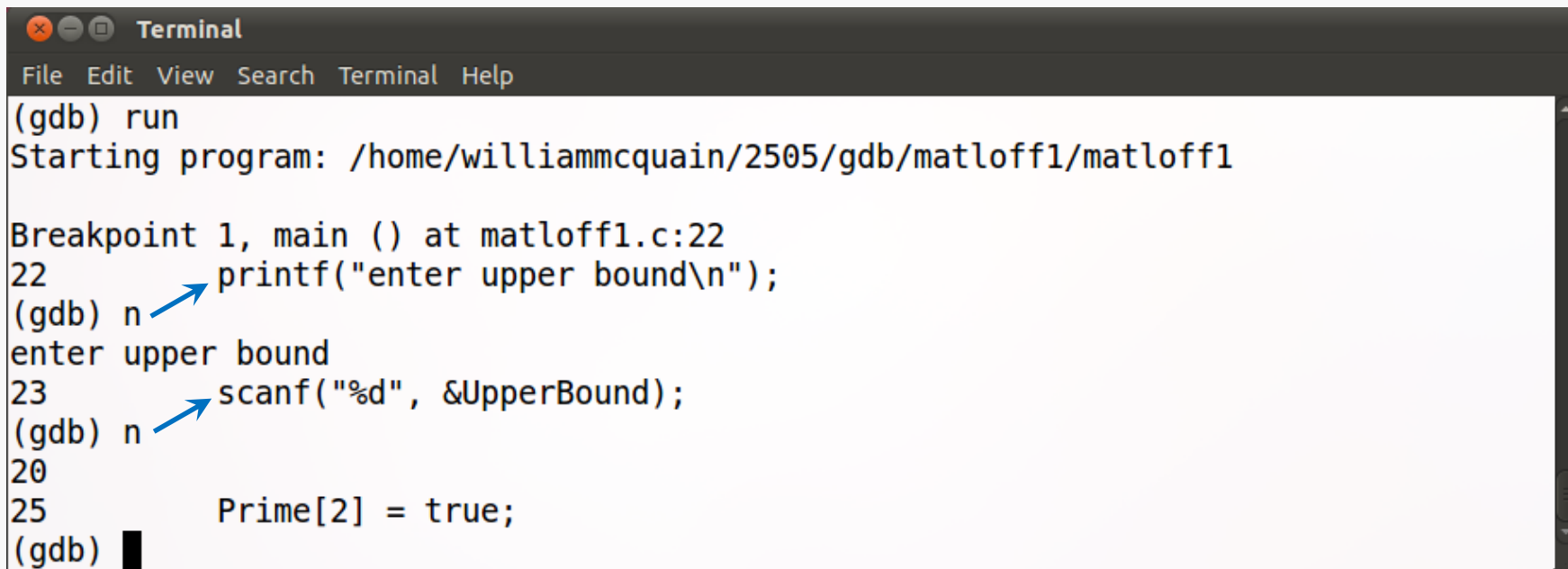
A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The session shows: "enter upper bound", "20", "Program exited normally.", "(gdb) break main", "Breakpoint 1 at 0x804844d: file matloff1.c, line 22.", "(gdb) run", "Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1", "Breakpoint 1, main () at matloff1.c:22", "22 printf(\"enter upper bound\\n\");", "(gdb)". A blue arrow points from the text below to the line "22 printf(\"enter upper bound\\n\");".

```
Terminal
File Edit View Search Terminal Help
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");
(gdb)
```

**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:

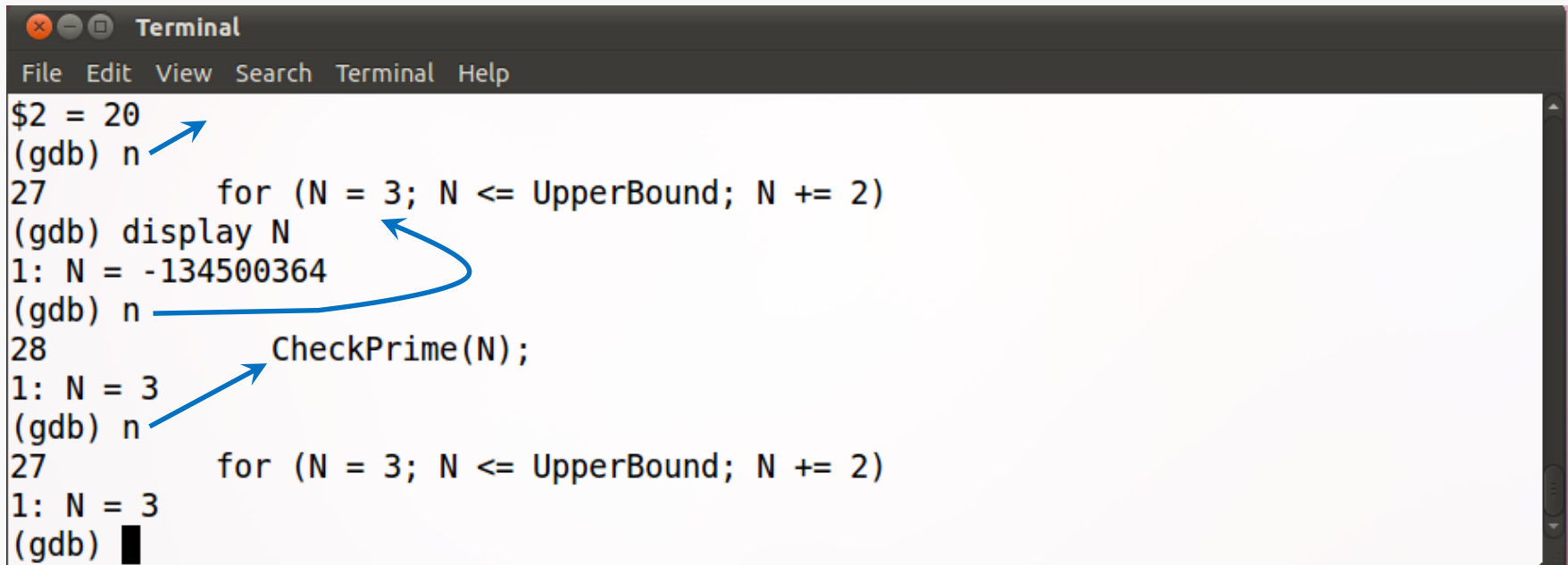
A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows a gdb session. It starts with "(gdb) run", followed by "Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1". A breakpoint is set at line 22 of matloff1.c: "Breakpoint 1, main () at matloff1.c:22". The terminal then shows line 22: "22 printf("enter upper bound\n");". A blue arrow points from the prompt "(gdb) n" to the first parameter of the printf call. The terminal then shows "enter upper bound". Next is line 23: "23 scanf("%d", &UpperBound);". A blue arrow points from the prompt "(gdb) n" to the first parameter of the scanf call. The terminal then shows "20" and "25 Prime[2] = true;". The prompt "(gdb) " is followed by a black cursor bar.

```
(gdb) run
Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1

Breakpoint 1, main () at matloff1.c:22
22 printf("enter upper bound\n");
(gdb) n
enter upper bound
23 scanf("%d", &UpperBound);
(gdb) n
20
25 Prime[2] = true;
(gdb)
```

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:



```
Terminal
File Edit View Search Terminal Help
$2 = 20
(gdb) n
27         for (N = 3; N <= UpperBound; N += 2)
(gdb) display N
1: N = -134500364
(gdb) n
28         CheckPrime(N);
1: N = 3
(gdb) n
27         for (N = 3; N <= UpperBound; N += 2)
1: N = 3
(gdb) █
```

The terminal screenshot shows a GDB session. The user sets a breakpoint at line 27 and steps through the code. The `display N` command is used to watch the value of variable `N`. The output shows that `N` is initially 3, but after stepping to line 28, it becomes -134500364. This indicates that the program has jumped to a different location, likely due to a loop or a branch. The user then steps back to line 27, and `N` is again 3.

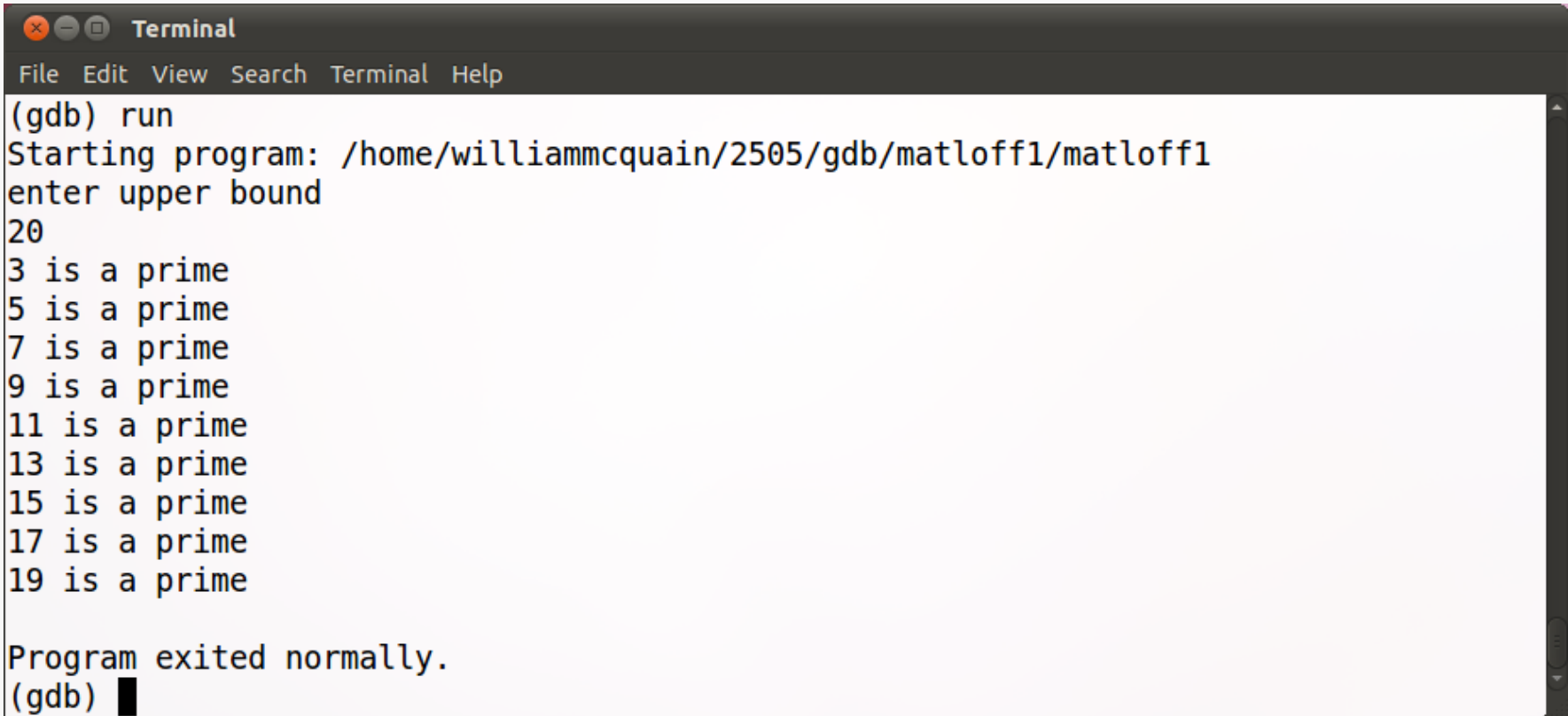
The initial display of `N` makes sense (why?), as does the next.

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()`).

Ah... missing braces around the intended body of the `for` loop:

```
. . .
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.



```
Terminal
File Edit View Search Terminal Help
(gdb) run
Starting program: /home/williammcquain/2505/gdb/matloff1/matloff1
enter upper bound
20
3 is a prime
5 is a prime
7 is a prime
9 is a prime
11 is a prime
13 is a prime
15 is a prime
17 is a prime
19 is a prime

Program exited normally.
(gdb) █
```

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...