

# Guide to Faster, Less Frustrating Debugging

**Norman Matloff**  
**University of California at Davis**  
**(530) 752-1953**  
**matloff@cs.ucdavis.edu**  
**©1992-2001, N. Matloff**

**April 4, 2002**

## Contents

- [1 Save Time and Frustration!](#)
- [2 General Debugging Strategies](#)
  - [2.1 Confirmation](#)
  - [2.2 Binary Search](#)
  - [2.3 What If It Doesn't Even Compile?](#)
- [3 Use a Debugging Tool and a Good Text Editor!](#)
  - [3.1 Don't Use printf\(\)/cout As Your Main Debugging Devices](#)
  - [3.2 Which Debugging Tools?](#)
    - [3.2.1 The Ubiquitous \(on Unix\) gdb](#)
    - [3.2.2 ddd: A Better View of gdb](#)
    - [3.2.3 A Good Text Editor Can Help a Lot](#)
  - [3.3 Integrated Development Environments](#)
  - [3.4 What Do I Use?](#)
- [4 How to Use gdb](#)
  - [4.1 Easy to Learn](#)
  - [4.2 The Basic Strategy](#)
  - [4.3 The Main gdb Commands](#)
    - [4.3.1 Invoking/Quitting gdb](#)
    - [4.3.2 The r \(Run\) Command](#)
    - [4.3.3 The l \(List\) Command](#)
    - [4.3.4 The b \(Breakpoint\) and c \(Continue\) Commands](#)
    - [4.3.5 The d \(Display\) and p \(Print\) Commands:](#)
    - [4.3.6 The printf Command](#)
    - [4.3.7 The n \(Next\) and s \(Step\) Commands](#)
    - [4.3.8 The bt \(Backtrace\) Command](#)
    - [4.3.9 The set Command](#)
  - [4.4 The call Command](#)
    - [4.4.1 The define Command](#)
  - [4.5 Effect of Recompiling the Source Without Exiting gdb](#)
  - [4.6 A gdb Example](#)
  - [4.7 Documentation on gdb](#)

## 1 Save Time and Frustration!

An integral part of your programming skills should be high proficiency in debugging. This document is intended as a first step toward that goal.

## 2 General Debugging Strategies

### 2.1 Confirmation

When your program contains a bug, it is of course because somewhere there is something which you believe to be true but actually is not true. In other words:

**Finding your bug is a process of confirming the many things you believe are true, until you find one which is not true.**

Here are examples of the types of things you might believe are true:

- You believe that at a certain point in your source file, a certain variable has a certain value.
- You believe that in a given if-then-else statement, the ``else" part is the one that is executed.
- You believe that when you call a certain function, the function receives its parameters correctly.

**So the processing of finding the location of a bug consists of confirming all these things!** If you believe that a certain variable should have a certain value at a certain point in time, check it! If you believe that the ``else" construct above is executed, check it!

Usually your belief will be confirmed, but eventually you will find a case where your belief will not be confirmed-and you will then know the location of the bug.

### 2.2 Binary Search

In this confirmation process, use a ``binary search" strategy. To explain this, suppose for the moment your program were one long file, say 200 lines long, with no function calls. (This would be terrible style, but again it will be easier to explain in this setting.)

Suppose you have an array `x`, and that you believe that `x[4] = 9` for almost the entire execution of the program. To check this, first check the value of `x[4]` at line 100. Say the value is 9. That means you have narrowed down the location of the bug to lines 101-200! Now check at line 150. Say there `x[4] = -127`, which is wrong. So, the location of the bug is now narrowed down to lines 101-150, and you will next check at line 125, and so on.

Of course, this is an oversimplified view, because hopefully your program does consist of function calls and thus we cannot simply divide numbers of lines by 2 in this manner, but you can see how you can quickly pinpoint the location of the bug but carefully choosing your checkpoints in something like a ``binary search" manner.

### 2.3 What If It Doesn't Even Compile?

Most compilation errors are obvious and easily fixed. But in some cases, you will just have no idea even where the error is. The compiler may tell you that the error is at the very last line of the definition of a function (or a class, in C++ or Java), even though all you have on that line is, say, ``}'`. That means the true location of the error could be anywhere in the function.

To deal with this, again use binary search! First, temporarily delete, say, the second half of the function. (Or, if done easily, comment-out that half.) You'll have to be careful in doing this, as otherwise you could introduce even more errors. If the error message disappears, then you know the problem is in that half; restore the deleted lines, and now delete half of that half, etc., until you pinpoint the location of the error.

## 3 Use a Debugging Tool and a Good Text Editor!

### 3.1 Don't Use printf()/cout As Your Main Debugging Devices

As seen above, the confirmation process often involves checking the values of variables at different points in your code and different points in time. In the past, you probably did this by adding temporary **printf** or **cout** statements to your C or C++ source file. This is very inconvenient: You have to recompile your program after you add them, and you may have to add a large number of them, say if you want to check the value of some variable at several different places in your program. Then after recompiling and running the program, you realize that there are even more places at which you need **printf/cout** statements, so you have to recompile again! Then after you fix that bug, you have to remove all these **printf/couts**, and start adding them to track down the next bug. Not only is this annoying, but it is distracting, making it difficult to concentrate, i.e. you will be apt to lose your train of thought on finding the bug.

A debugging tool allows you to print out values of variables much more conveniently: You don't have to recompile, and you can ask the debugging tool to monitor any variables you like, automatically. Again, the "convenience factor" here is quite important. The less time you spend on recompiles, etc., the more time-and mental energy-you have available to concentrate on hunting down the bug.

**Even more important, the debugging tool will tell you where in your program a disastrous error (e.g. a "segmentation fault") occurs. In addition, it can tell you where the function generating the seg fault was called from. This is extremely useful.**

There are times when **printf/couts** are useful in conjunction with a debugging tool, but you'll be doing yourself a favor if you use the debugging tool, not **printf/couts**, as your main debugging device.

### 3.2 Which Debugging Tools?

#### 3.2.1 The Ubiquitous (on Unix) gdb

Most computer systems have one or more debugging tools available. These can save you a tremendous amount of time and frustration in the debugging process. The tool available on almost all Unix systems is **gdb**.

Of course, once you become adept at using one debugging tool, it will be very simple to learn others. So, although we use **gdb** as our example here, the principles apply to any debugging tool. (I am not saying that **gdb** is the best debugger around, but since it is so common I am using it as my example.)

#### 3.2.2 ddd: A Better View of gdb

This is a text-based tool, and a number of GUI "front ends" have been developed for it, such as **ddd**. **I strongly recommend using a GUI-based interface like this for gdb**; see my debugging home page for how to obtain and use these: <http://heather.cs.ucdavis.edu/~matloff/debug.html>

It's much easier and pleasurable to use **gdb** through the **ddd** interface. For example, to set a breakpoint we just click the mouse, and a little stop sign appears to show that we will stop there.

But I do recommend learning the text-based version first.

#### 3.2.3 A Good Text Editor Can Help a Lot

During a long debugging session, you are going to spend a lot of time just typing. Not only does this waste precious time, but much more importantly, it is highly distracting; it's easy to lose your train of thought while typing, especially when you must frequently hop around from one part of a file to another, or between files.

I have written some tips for text editing, designed specifically for programmers, at <http://heather.cs.ucdavis.edu/~matloff/progedit.html>. For example, it mentions that you should make good use of undo/redo operations.<sup>1</sup> Consider our binary-search example above, in which we were trying to find an elusive compilation error. The advice given was to delete half the lines in the function, and later restore them. If your text editor includes undo capabilities, then the restoration of those deleted lines will be easy.

It's very important that you use an editor which allows subwindows. This enables you to, for instance, look at the definition of a function in one window while viewing a call to the function in another window.

Often one uses other tools in conjunction with a debugger. For example, the **vim** editor (an enhanced version of **vi**) can interface with **gdb**; see my **vim** Web page: <http://heather.cs.ucdavis.edu/~matloff/vim.html>. You can initiate a compile from within **vim**, and then if there are compilation errors, **vim** will take you to the lines at which they occur.

### 3.3 Integrated Development Environments

In addition, a debugger will sometimes be part of an overall package, known as an *integrated development environment* (IDE). An example of the many IDEs available for Unix is Code Crusader: <http://heather.cs.ucdavis.edu/~matloff/codecrusader.html>

It used to be public-domain, but unfortunately Code Crusader is about to become a commercial product.

One big drawback from the point of view of many people is that one cannot use one's own text editor in most IDEs. Many people would like to use their own editor for everything-programming, composing e-mail, word processing and so on. This is both convenient and also allows them to develop personal alias/macro libraries which save them work.

A big advantage of Code Crusader is that it allows you to use your own text editor. As far as I can tell, this works best with **emacs**.

### 3.4 What Do I Use?

Mostly I use **gdb**, accessed from **ddd**, along with **vim** for my editor. I occasionally use Code Crusader.

## 4 How to Use gdb

### 4.1 Easy to Learn

In my own debugging, I tend to use just a few **gdb** commands, only four or five in all. So, you can learn **gdb** quite quickly. Later, if you wish, you can learn some advanced commands.

### 4.2 The Basic Strategy

A typical usage of **gdb** runs as follows: After starting up **gdb**, we set *breakpoints*, which are places in the code where we wish execution to pause. Each time **gdb** encounters a breakpoint, it suspends execution of the program at that point, giving us a chance to check the values of various variables.

In some cases, when we reach a breakpoint, we will *single step* for a while from that point onward, which means that **gdb** will pause after every line of source code. This may be important, either to further pinpoint the location at which a certain variable changes value, or in some cases to observe the flow of execution, seeing for example which parts of if-then-else constructs are executed.

**As mentioned earlier, another component of the overall strategy concerns segmentation faults.** If we receive a "seg fault" error message when we execute our program (running by itself, not under **gdb**), we can then run the program under **gdb** (probably not setting any breakpoints at all). When the seg fault occurs, **gdb** will tell us exactly where it happened, again pinpointing the location of the error. (In some cases the seg fault itself will occur within a system call, rather than in a function we wrote, in which case **gdb**'s **bt** command can be used to determine where in our code we made the system call.) At that point you should check the values of all array indices and pointers which are referenced in the line at which the error occurred. Typically you will find that either you have an array index with a value far out of range, or a pointer which is 0 (and thus unreferenceable). Another common error is forgetting an ampersand in a function call, say `scanf()`. Still another one is that you have made a system call which failed, but you did not check its return value for an error code.

## 4.3 The Main gdb Commands

### 4.3.1 Invoking/Quitting gdb

Before you start, make sure that when you compiled the program you are debugging, you used the `-g` option, i.e.

```
cc -g sourcefile.c
```

Without the `-g` option, **gdb** would essentially be useless, since it will have no information on variable and function names, line numbers, and so on.

Then to start **gdb** type

```
gdb filename
```

where 'filename' is the executable file, e.g. `a.out` for your program.

To quit **gdb**, type `q`.

### 4.3.2 The r (Run) Command

This begins execution of your program. Be sure to include any command-line arguments; e.g. if in an ordinary (i.e. nondebugging) run of your program you would type

```
a.out < z
```

then within **gdb** you would type

```
r < z
```

If you apply **r** more than once in the same debugging session, you do not have to type the command-line arguments after the first time; the old ones will be repeated by default.

### 4.3.3 The **l** (List) Command

You can use this to list parts of your source file(s). E.g. typing

```
l 52
```

will result in display of Line 52 and the few lines following it (to see more lines, hit the carriage return again).

If you have more than one source file, precede the line number by the file name and a colon, e.g.

```
l x.c:52
```

You can also specify a function name here, in which case the listing will begin at the first line of the function.

The **l** command is useful to find places in the file at which you wish to set breakpoints (see below).

### 4.3.4 The **b** (Breakpoint) and **c** (Continue) Commands

This says that you wish execution of the program to pause at the specified line. For example,

```
b 30
```

means that you wish to stop every time the program gets to Line 30.

Again, if you have more than one source file, precede the line number by the file name and a colon as shown above.

Once you have paused at the indicated line and wish to continue executing the program, you can type **c** (for the **continue** command).

You can also use a function name to specify a breakpoint, meaning the first executable line in the function. For example,

```
b main
```

says to stop at the first line of the main program, which is often useful as the first step in debugging.

You can cancel a breakpoint by using the **disable** command.

You can also make a breakpoint conditional. E.g.

```
b 3 z > 92
```

would tell **gdb** to stop at breakpoint 3 (which was set previously) only when Z exceeds 92.

### 4.3.5 The **d** (Display) and **p** (Print) Commands:

This prints out the value of the indicated variable or expression every time the program pauses (e.g. at breakpoints and after executions of the **n** and **s** commands). E.g. typing

```
disp NC
```

once will mean that the current value of the variable NC will automatically be printed to the screen every time the program pauses.

If we have **gdb** print out a **struct** variable, the individual fields of the struct will be printed out. If we specify an array name, the entire array will be printed.

After a while, you may find that the displaying a given variable or expression becomes less valuable than it was before. If so, you can cancel a **disp** command by using the **undisplay** command.

A related command is **p**; this prints out the value of the variable or expression just once.

In both cases, keep in mind the difference between global and local variables. If for example, you have a local variable L within the function F, then if you type

```
disp L
```

when you are not in F, you will get an error message like ``No variable L in the present context."

**gdb** also gives you the option of using nondefault formats for printing out variables with **disp** or **p**. For example, suppose you have declared the variable G as

```
int G;
```

Then

```
p G
```

will result in printing out the variable in integer format, like

```
printf("%d\n",G);
```

would. But you might want it in hex format, for example, i.e. you may wish to do something like

```
printf("%x\n",G);
```

**gdb** allows you to do this by typing

```
p /x G
```

### 4.3.6 The printf Command

This is even better, as it works like C's function of the same name. For example, suppose you have two integer variables, X and Y, which you would like to have printed out. You can give **gdb** the command:

```
printf "X = %d, Y = %d\n",X,Y
```

### 4.3.7 The n (Next) and s (Step) Commands

These tell **gdb** to execute the next line of the program, and then pause again. If that line happens to be a function call, then **n** and **s** will give different results:

If you use **s**, then the next pause will be at the first line of the function; if you use **n**, then the next pause will be at the line *following the function call* (the function will be single-step executed, but there will be no pauses within it). This is very important, and can save you a lot of time: If you think the bug does not lie within the function, then use **n**, so that you don't waste a lot of time single-stepping within the function itself.

When you use **s** at a function call, **gdb** will also tell you the values of the parameters, which is useful for confirmation purposes, as explained at the beginning of this document.

### 4.3.8 The bt (Backtrace) Command

If you have an execution error with a mysterious message like ``bus error" or ``segmentation fault," the **bt** command will at least tell you where in your program this occurred, and if in a function, where the function was called from. **This can be extremely valuable information.**

### 4.3.9 The set Command

Sometimes it is very useful to use **gdb** to change the value of a program variable, and this command will do this. For example, if you have an **int** variable x, the **gdb** command

```
set variable x = 12
```

will change x's value to 12.

## 4.4 The call Command

You can use this function to call a function in your program during execution. Typically you do this with a function which you've written for debugging purposes, e.g. to print out a linked list.

Example:

```
(gdb) call x()
```

Make sure to remember to type the parentheses, even if there are no arguments.



### 4.4.1 The define Command

This saves you typing. You can put together one or more commands into a macro. For instance, recall our example from above,

```
printf "X = %d, Y = %d\n",X,Y
```

If you wanted to frequently use this command during your debugging session, you could do:

```
(gdb) define pxy
Type commands for definition of "pxy".
End with a line saying just "end".
>printf "%X = %d, Y = %d\n",X,Y
>end
```

Then you could invoke it just by typing ``pxy".

## 4.5 Effect of Recompiling the Source Without Exiting gdb

As you know, a debugging session consists of compiling, then debugging, then editing, then recompiling, then debugging, then editing, then recompiling...

A key point is that you should not exit **gdb** before recompiling. After recompiling, when you issue the **r** command to rerun your program, **gdb** will notice that the source file is now newer than the binary executable file which it had been using, and thus will automatically reload the new binary before the rerun. Since it takes time to start **gdb** from scratch, it's much easier to stay in **gdb** between compiles, rather than exiting and then starting it up again.

## 4.6 A gdb Example

In this section we will introduce **gdb** by showing a **script** file record of its use on an actual program. In order to distinguish between line numbers in the **script** file from line numbers within the C source files, I have placed a `g' at the beginning of each of the former. For example, Line g56 means Line 56 within the **script** file, not Line 56 within one of the C source files.)

First, use **cat** to show the program source files:

```
g2  mole.matloff% cat Main.c
g3
g4
g5  /* prime-number finding program
g6
g7      will (after bugs are fixed) report a list of all primes which are
g8      less than or equal to the user-supplied upper bound
g9
g10     riddled with errors! */
g11
g12
g13
g14  #include "Defs.h"
g15
g16
```

```

g17 int Prime[MaxPrimes], /* Prime[I] will be 1 if I is prime, 0 otherwi */
g18     UpperBound; /* we will check all number up through this one for
g19                 primeness */
g20
g21
g22 main()
g23
g24 {   int N;
g25
g26     printf("enter upper bound\n");
g27     scanf("%d",UpperBound);
g28
g29     Prime[2] = 1;
g30
g31     for (N = 3; N <= UpperBound; N += 2)
g32         CheckPrime();
g33         if (Prime[N]) printf("%d is a prime\n",N);
g34 }
g35
g36 mole.matloff%
g37 mole.matloff%
g38 mole.matloff%
g39 mole.matloff% cat CheckPrimes.c
g40 cat: CheckPrimes.c: No such file or directory
g41 mole.matloff% cat CheckPrime.c
g42
g43
g44 #include "Defs.h"
g45 #include "Externs.h"
g46
g47
g48 CheckPrime(K)
g49     int K;
g50
g51 {   int J;
g52
g53     /* the plan:  see if J divides K, for all values J which are
g54
g55         (a) themselves prime (no need to try J if it is nonprime), and
g56         (b) less than or equal to sqrt(K) (if K has a divisor larger
g57         than this square root, it must also have a smaller one,
g58         so no need to check for larger ones) */
g59
g60     J = 2;
g61     while (1)  {
g62         if (Prime[J] == 1)
g63             if (K % J == 0)  {
g64                 Prime[K] = 0;
g65                 return;
g66             }
g67         J++;
g68     }
g69
g70     /* if we get here, then there were no divisors of K, so it is
g71     prime */
g72     Prime[K] = 1;
g73 }
g74
g75 mole.matloff%
g76 mole.matloff%
g77 mole.matloff%
g78 mole.matloff% cat Defs.h
g79
g80 #define MaxPrimes 50
g81 mole.matloff% cat Externs.h

```

```

g82
g83
g84 #include "Defs.h"
g85
g86
g87 extern int Prime[MaxPrimes];

```

The comments in Lines g5-g10 state what the program goal is, i.e. finding prime numbers.

OK, let's get started. First we compile the program:

```

g92 mole.matloff% make
g93 cc -g -c Main.c
g94 cc -g -c CheckPrime.c
g95 "CheckPrime.c", line 31: warning: statement not reached
g96 cc -g -o FindPrimes Main.o CheckPrime.o

```

The warning concerning Line 31 of CheckPrime.c sounds ominous (and it *is*), but let's ignore it, and see what happens. Let's run the program:

```

g100 mole.matloff% FindPrimes
g101 enter upper bound
g102 20
g103 Segmentation fault

```

Well, this sounds scary, but actually it usually is the easiest type of bug to fix. The first step is to determine where the error occurred; **gdb** will do this for us: We enter **gdb** and then re-run the program, so as to reproduce the error:

```

g104 mole.matloff% gdb FindPrimes
g105 GDB is free software and you are welcome to distribute copies of it
g106 under certain conditions; type "show copying" to see the conditions.
g107 There is absolutely no warranty for GDB; type "show warranty" for details
g108 GDB 4.7, Copyright 1992 Free Software Foundation, Inc...

```

OK, **gdb** is now ready for my command (it indicates this by giving me a special prompt, which looks like this:

```
(gdb)
```

I now invoke **gdb**'s **r** command, to run the program (if the program had any command-line arguments, I would have typed them right after the **`r'**):

```

g109 (gdb) r
g110 Starting program: /tmp_mnt/lion/d/guest/matloff/tmp/FindPrimes
g111 enter upper bound
g112 20
g113
g114 Program received signal 11, Segmentation fault
g115 0xf773cb88 in _doscan ( )

```

So, the error occurred within the function `_doscan()`. This is not one of my functions, so it must have been called by one of the C library functions which I am using, i.e. `printf()` or `scanf()`. Given the name `_doscan`, it does sound like it must have been the latter, but at any rate, the way to find out is to use **gdb's** **bt** ("backtrace") command, to see where `_doscan()` was called from:

```
g116  (gdb) bt
g117  #0  0xf773cb88 in _doscan ()
g118  #1  0xf773c2e8 in _doscan ()
g119  #2  0xf773b9dc in scanf ()
g120  #3  0x22dc in main () at Main.c:25
```

Aha! So it was indeed called from `scanf()`, which in turn was called from `main()`, at Line 25.

Now since `scanf()` is a C library function, it presumably is well debugged already, so the error was probably not in `scanf()`. So, the error must have been in our call to `scanf()` on Line 25 of `Main.c`.

Let's take a look at that latter call. To do so, we use **gdb's** **l** ("list") command, to list some lines of the file `Main.c` at and around the line which led to the error:

```
g121  (gdb) l Main.c:25
g122  20      main()
g123  21
g124  22      {  int N;
g125  23
g126  24          printf("enter upper bound\n");
g127  25          scanf("%d",UpperBound);
g128  26
g129  27          Prime[2] = 1;
g130  28
g131  29          for (N = 3; N <= UpperBound; N += 2)
```

Yep, a famous "C-learner's error"-we forgot the ampersand in before `UpperBound`! Line 25 of `Main.c` should have been

```
scanf("%d",&UpperBound);
```

So, in another window (hopefully an X11 window, but if you are just using a text-based terminal, you can use the GNU **screen** program to get windows), we will fix line 25 of `Main.c`, and recompile. Note that we do not leave **gdb** while doing this, since **gdb** takes a long time to load. In order to do this, though, we must first tell **gdb** to relinquish our executable file:

```
(gdb) kill
```

(Otherwise when we tried to recompile our program, the **ld** linker would tell us that the executable file is "busy" and thus cannot be replaced.)

After fixing and recompiling `Main.c`, the next time we give **gdb** the `run` command

```
(gdb) r
```

**gdb** will automatically load the newly-recompiled executable for our program (it will notice that we recompiled, because it will see that our .c source file is newer than the executable file). Note that we do not have to state the command-line arguments (if any), because **gdb** remembers them from before. It also remembers our breakpoints, so we do not have to set them again. (And **gdb** will automatically update the line numbers for those breakpoints, adjusting for whatever line-number changes occurred when we modified the source file.)

Main.c is now the following:

```

g137 mole.matloff% cat Main.c
g138
g139
g140 /* prime-number finding program
g141
g142     will (after bugs are fixed) report a list of all primes which are
g143     less than or equal to the user-supplied upper bound
g144
g145     riddled with errors! */
g146
g147
g148
g149 #include "Defs.h"
g150
g151
g152 int Prime[MaxPrimes], /* Prime[I] will be 1 if I is prime, 0 otherwi */
g153     UpperBound; /* we will check all number up through this one for
g154                 primeness */
g155
g156
g157 main()
g158 {   int N;
g159
g160     printf("enter upper bound\n");
g161     scanf("%d",&UpperBound);
g162
g163     Prime[2] = 1;
g164
g165     for (N = 3; N <= UpperBound; N += 2)
g166         CheckPrime();
g167         if (Prime[N]) printf("%d is a prime\n",N);
g168
g169 }
```

Now we run the program again (not in **gdb**, though we do still have **gdb** open in the other window):

```

g174 mole.matloff% !F
g175 FindPrimes
g176 enter upper bound
g177 20
g178 Segmentation fault
```

Don't get discouraged! Let's see where this new seg fault is occurring.

```

g188 (gdb) r
g189 Starting program: /tmp_mnt/lion/d/guest/matloff/tmp/FindPrimes
g190 enter upper bound
g191 20
g192
g193 Program received signal 11, Segmentation fault
g194 0x2388 in CheckPrime (K=1) at CheckPrime.c:21
```

```
g195 21          if (Prime[J] == 1)
```

Now, remember, as mentioned earlier, one of the most common causes of a seg fault is a wildly-erroneous array index. Thus we should be highly suspicious of J in this case, and should check what its value is, using **gdb's** **p** (``print'`) command:

```
g207 (gdb) p J
g208 $1 = 4024
```

Wow! Remember, I only had set up the array Prime to contain 50 integers, and yet here we are trying to access Prime[4024]!<sup>2</sup>

So, **gdb** has pinpointed the exact source of our error-the value of J is way too large on this line. Now we have to determine why J was so big. Let's take a look at the entire function, using **gdb's** **l** command:

```
g196 (gdb) l CheckPrime.c:12
g53  /* the plan:  see if J divides K, for all values J which are
g54
g55      (a) themselves prime (no need to try J if it is nonprime), and
g56      (b) less than or equal to sqrt(K) (if K has a divisor larger
g57      than sqrt(K), it must also have a smaller one,
g58      so no need to check for larger ones) */
g59
g200 19      J = 2;
g201 20      while (1) {
g202 21          if (Prime[J] == 1)
g203 22              if (K % J == 0) {
g204 23                  Prime[K] = 0;
g205 24                  return;
g206 25              }
```

Look at the comments in Lines g56-g58. We were supposed to stop searching after J got to sqrt(K). Yet you can see in Lines g201-g206 that we never made this check, so J just kept growing and growing, eventually reaching the value 4024 which triggered the seg fault.

After fixing this problem, the new CheckPrime.c looks like this:

```
g214 mole.matloff% cat CheckPrime.c
g215
g216
g217 #include "Defs.h"
g218 #include "Externs.h"
g219
g220
g221 CheckPrime(K)
g222     int K;
g223
g224 {     int J;
g225
g226     /* the plan:  see if J divides K, for all values J which are
g227
g228         (a) themselves prime (no need to try J if it is nonprime), and
g229         (b) less than or equal to sqrt(K) (if K has a divisor larger
g230         than this square root, it must also have a smaller one,
g231         so no need to check for larger ones) */
```

```

g232
g233     for (J = 2; J*J <= K; J++)
g234         if (Prime[J] == 1)
g235             if (K % J == 0) {
g236                 Prime[K] = 0;
g237                 return;
g238             }
g239
g240     /* if we get here, then there were no divisors of K, so it is
g241        prime */
g242     Prime[K] = 1;
g243 }

```

OK, let's give it another try:

```

g248 mole.matloff% !F
g249 FindPrimes
g250 enter upper bound
g251 20
g252 mole.matloff%

```

What?! No primes reported up to the number 20? That's not right. Let's use **gdb** to step through the program. We will pause at the beginning of `main()`, and take a look around. To do that, we set up a "breakpoint," i.e. a place where **gdb** will suspend execution of our program, so that we can assess the situation before resuming execution:

```

g261 (gdb) b main
g262 Breakpoint 1 at 0x22b4: file Main.c, line 24.

```

So, **gdb** will pause execution of our program whenever it hits Line 24 of the file `Main.c`. This is Breakpoint 1; we might (and will) set other breakpoints later, so we need numbers to distinguish them, e.g. in order to specify which one we want to cancel.

Now let's run the program, using the **r** command:

```

g286 (gdb) r
g287 Starting program: /tmp_mnt/lion/d/guest/matloff/tmp/FindPrimes
g288
g289 Breakpoint 1, main () at Main.c:24
g290 24      printf("enter upper bound\n");

```

We see that, as planned, **gdb** did stop at the first line of `main()` (Line 24). Now we will execute the program one line at a time, using **gdb's** **n** ("next") command:

```

g291 (gdb) n
g292 enter upper bound
g293 25      scanf("%d", &UpperBound);

```

What happened was that **gdb** executed Line 24 of `Main.c` as requested-the message from the call to `printf()` appears on Line g292-and now has paused again, at Line 25 of `Main.c`, displaying that line for us (Line g293 of the **script** file).

OK, let's execute Line 25, by typing ``n'` again:

```
g294 (gdb) n
g295 20
g296 27          Prime[2] = 1;
```

Since Line 25 was a `scanf()` call, at Line g295 of the **script** file, **gdb** waited for our input, which we typed as 20. **Gdb** then executed the `scanf()` call, and paused again, now at Line 27 of `Main.c` (Line 296 of the **script** file).

Now let's check to make sure that `UpperBound` was read in correctly. We think it was, but remember, the basic principle of debugging is to check anyway. To do this, we will use **gdb's** **p** (``print'`) command:

```
g297 (gdb) p UpperBound
g298 $1 = 20
```

OK, that's fine. So, let's continue to execute the program one line at a time, by using **n**:

```
g299 (gdb) n
g300 29          for (N = 3; N <= UpperBound; N += 2)
```

Also, let's keep track of the value of `N`, using **gdb's** **disp** (``display'`) command. The latter is just like the **p**, except that **disp** will print out the value of the variable each time the program pauses, as opposed to **p**, which prints out the value only once.

```
g301 (gdb) disp N
g302 1: N = 0
g303 (gdb) n
g304 30          CheckPrime();
g305 1: N = 3
g306 (gdb) n
g307 29          for (N = 3; N <= UpperBound; N += 2)
g308 1: N = 3
```

Hey, what's going on here? After executing Line 30, the program then went back to Line 29-skipping Line 31. Here is what the loop looked like:

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

Oops! We forgot the braces. Thus only Line 30, not Lines 30 and 31, forms the body of the loop. No wonder Line 31 wasn't executed.

After fixing that, `Main.c` looks like this:

```
g314 mole.matloff% cat Main.c
g315
```



```

g316
g317 /* prime-number finding program
g318
g319 will (after bugs are fixed) report a list of all primes which are
g320 less than or equal to the user-supplied upper bound
g321
g322 riddled with errors! */
g323
g324
g325
g326 #include "Defs.h"
g327
g328
g329 int Prime[MaxPrimes], /* Prime[I] will be 1 if I is prime, 0
                        otherwise */
g330     UpperBound; /* we will check all number up through this one for
g331                  primeness */
g332
g333
g334 main()
g335 {
g336     int N;
g337
g338     printf("enter upper bound\n");
g339     scanf("%d",&UpperBound);
g340
g341     Prime[2] = 1;
g342
g343     for (N = 3; N <= UpperBound; N += 2) {
g344         CheckPrime();
g345         if (Prime[N]) printf("%d is a prime\n",N);
g346     }
g347 }

```

OK, try again:

```

g352 mole.matloff% !F
g353 FindPrimes
g354 enter upper bound
g355 20
g356 mole.matloff%

```

Still no output! Well, we will now need to try a more detailed line-by-line execution of the program. Last time, we did not go through the function CheckPrime() line-by-line, so we will need to now:

```

g586 (gdb) l Main.c:1
g587 1
g588 2
g589 3      /* prime-number finding program
g590 4
g591 5      will (after bugs are fixed) report a list of all primes which
g592 (gdb)      are less than or equal to the user-supplied upper bound
g593 6
g594 7      riddled with errors! */
g595 8
g596 9
g597 10
g598 11
g599 12     #include "Defs.h"
g600 13

```

```

g601 14
g602 15      int Prime[MaxPrimes], /* Prime[I] will be 1 if I is prime, 0
g603 (gdb)
g604 16      UpperBound; /* we will check all number up through this one
g605 17                  primeness */
g606 18
g607 19
g608 20      main()
g609 21
g610 22      {  int N;
g611 23
g612 24          printf("enter upper bound\n");
g613 25          scanf("%d",&UpperBound);
g614 (gdb)
g615 26
g616 27          Prime[2] = 1;
g617 28
g618 29          for (N = 3; N <= UpperBound; N += 2)  {
g619 30              CheckPrime();
g620 31              if (Prime[N]) printf("%d is a prime\n",N);
g621 32          }
g622 33      }
g623 34
g624 (gdb) b 30
g625 Breakpoint 1 at 0x2308: file Main.c, line 30.

```

Here we have placed a breakpoint at the call to `CheckPrime`.<sup>3</sup>

Now, let's run the program:

```

g626 (gdb) r
g627 Starting program: /tmp_mnt/lion/d/guest/matloff/tmp/FindPrimes
g628 enter upper bound
g629 20
g630
g631 Breakpoint 1, main () at Main.c:30
g632 30      CheckPrime();

```

**Gdb** has stopped at Line 30, as we requested. Now, instead of using the **n** command, we will use **s** (``step'`). This latter command is the same as **n**, except that it will enter the function rather than skipping over the function like **n** does:

```

g633 (gdb) s
g634 CheckPrime (K=1) at CheckPrime.c:19
g635 19      for (J = 2; J*J <= K; J++)

```

Sure enough, **s** has gotten us to the first line within `CheckPrime()`.

Another service **gdb** provides for us is to tell us what the values of the parameters of the function are, in this case  $K = 1$ . But that doesn't sound right—we shouldn't be checking the number 1 for primeness. So **gdb** has uncovered another bug for us.

In fact, our plan was to check the numbers 3 through `UpperBound` for primeness: The **for** loop in `main()` had the following heading:

```
for (N = 3; N <= UpperBound; N += 2)
```

Well, what about the call to `CheckPrime()`? Here is the whole loop from `main()`:

```

29         for (N = 3; N <= UpperBound; N += 2)  {
30             CheckPrime();
31             if (Prime[N]) printf("%d is a prime\n",N);
32         }

```

Look at Line 30-we forgot the parameter! This line should have been

```

30             CheckPrime(N);

```

After fixing this, try running the program again:

```

g699  mole.matloff% !F
g700  FindPrimes
g701  enter upper bound
g702  20
g703  3 is a prime
g704  5 is a prime
g705  7 is a prime
g706  11 is a prime
g707  13 is a prime
g708  17 is a prime
g709  19 is a prime

```

OK, the program now seems to be working.

## 4.7 Documentation on gdb

There is a bound manual that you can buy, but it is probably not worthwhile, since you can get all the documentation online. First you can get an overview of the categories of commands by simply typing ``h'` (``help'`), and then get information on any category or individual command by typing ``h name'`, where ``name'` is the name of the category or individual command. For example, to get information on using breakpoints, type

```
h breakpoints
```

---

### Footnotes:

<sup>1</sup>That in turns means that you should make sure to use a good editor which has these operations.

<sup>2</sup>Note very, very carefully, though: Most C compilers-including the Unix one, which is what we are using here-do not produce checks for violation of array bounds. In fact, a ```moderate''` violation, e.g. trying to access `Prime[57]`, would not have produced a seg fault. The reason that our attempt to access `Prime[4024]` did produce a seg fault is that it resulted in our trying to access memory which did not belong to us, i.e. belonged to some other user of the machine. The virtual-memory hardware of the machine detected this.

By the way, the ``$1'` here just means that we can refer to 4024 by this name from now on if we like.

<sup>3</sup>We could have simply typed ```b CheckPrime"`, which would have set a breakpoint at the first line of `CheckPrime()`, but doing it this way gives us a chance to see how the `s` command works. By the way, note Lines 592, 603 and 614; here we simply typed the carriage return, which results in **gdb** listing some further lines for us.

---

File translated from T<sub>E</sub>X by [T<sub>T</sub>H](#), version 2.88.

On 4 Apr 2002, 16:47.