An Interactive Guide to Faster, Less Frustrating Debugging
Adapted from http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html
by Norman Matloff

Contents:

Introduction

An integral part of your programming skills should be high proficiency in debugging. This document is intended as a first step toward that goal. Since this is an adaptation of Professor Matloff's tutorial, all mention of "I" in the text refers to him.

General Debugging Strategies

Confirmation

When your program contains a bug, it is of course because somewhere there is something that 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 that 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 process 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.

Binary Search

Much of the task of debugging is finding the location of an error. Of course you could confirm every line of code in your program, but this would be tedious and waste your time when there is a better way. You should confirm your program using 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 this strategy 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.

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 comment-out the second half of the function. 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 commented lines. Now comment-out half of that half, etc., until you pinpoint the location of the error.

How to Use gdb

Preparing for the Interactive Part of the Tutorial

For this tutorial you will need to copy two files from the ~davis/gdb directory. Create an empty directory, and then use cd to move into this empty directory. From now on, we will print in **bold** the commands you should type. We will use the `Courier Font` to indicate output on the screen. Now type (note the final period to indicate that the current directory is the destination):

cp ~ssdavis/gdb/*.c .

If you type **ls**, you should see:

```
main.c  prime.c
```

You will find a copy of both files on the last page of this handout. Please tear off the last page so you can easily reference line numbers when they are mentioned. As indicated in the comment at the top of main.c, this program is designed to find the prime numbers between 1 and the number entered by the user.

When you wish to use gdb with your executable program, you must make sure that you compile using the –g option, e.g., gcc -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. Now type:

gcc –g main.c prime.c

You should use ls to ensure that you now have a.out, the executable. Now try running the program by typing:

a.out (if you do not have the current directory in your path, then you will have to type **./a.out**)

Enter **20** at the prompt. After you have entered a number, the program should cause a 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 can do this in two ways. One way is to enter gdb and then re-run the program, so as to reproduce the error. The other way involves using a diagnostic file produced when a program terminates improperly. If you type **ls** you will find that there is a file named "core" now in your directory. We will see shortly how to use this core file to help in our debugging. It is now time to use gdb.

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.

It is wise to have two windows open when debugging: one in which to edit source, and one in which to run gdb. You should now open that second window in which to run gdb. After opening the window, change into the directory in which you copied main.c and prime.c.

gdb Course assignments

If you are doing this tutorial for a course, we will want a record of your work with gdb. The *script* program copies all output to the monitor into a file called *typescript*. When you are done with the tutorial, you terminate the *script* program by typing *exit*, and have the *typescript* file to handin.

In the window in which you will run gdb, now type **script.** You will get the message, "`Script started, file is typescript`."

This tutorial will tell you when to terminate the *script* program. Do not worry about doing the tutorial perfectly for the script!! All we care about is that you did the tutorial, not if you made mistakes along the way.

The Main gdb Commands

Invoking gdb

To start gdb you type gdb followed by the executable filename. Now type:

gdb a.out

The h (Help) Command

The commands are not case sensitive. Command abbreviations are allowed as long as they are not ambiguous. If you have any questions about gdb, the *help* command should be your first resort. Take a look at gdb's help before continuing by typing:

h

The r (Run) Command

This command begins execution of your program. If your executable normally takes command-line arguments (the prime program does not), then you must include the command line arguments with the *run* command. For example, if in an ordinary run of your program you would type "a.out param param2" then within gdb you would type "r param param2". If you apply *run* 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. Now type:

r

You should now see the prompt for the program. Enter **20** at the prompt. You should then see something similar to the following:

```
Enter upper bound:
20

Program received signal SIGSEGV, Segmentation fault.
0x4006af3b in _IO_vfscanf (s=0x401098a0, format=0x8048565 "%d",
    argptr=0xbffffae0, errp=0x0) at vfscanf.c:963
963     vfscanf.c: No such file or directory.
(gdb)
```

So, the error occurred within the function _IO_vfscanf(). 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 IO_vcanf, it does sound like it must have been the latter. We can use gdb's *Backtrace* command, to see from where IO_vscanf() was called.

The bt (Backtrace) Command

If you have an execution error with a mysterious message like ``bus error" or ``segmentation fault," the *Backtrace* command will at least tell you where in your program this occurred, and if in a function, where the function was called from. Since locating the error is often the key to solving a bug, this can be extremely valuable information. Now type:

bt

You should see something similar to the following:

```
#0  0x4006af3b in _IO_vfscanf (s=0x401098a0, format=0x8048565 "%d",
    argptr=0xbffffaf0, errp=0x0) at vfscanf.c:963
#1  0x4006c8aa in scanf (format=0x8048565 "%d") at scanf.c:33
#2  0x8048423 in main () at main.c:17
(gdb)
```

If you look at the back page you tore off earlier, you will see that line #17 of main.c is:

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

Aha! So it was indeed called from scanf(), which in turn was called from main(), at Line #17. 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 17 of main.c.

Before we continue, we can demonstrate another way to get to the same information. First, we need to terminate our debugging session using the gdb *Kill* command and then confirm that we wish to kill the program. Now type:

k
y

Now quit gdb using the *Quit* command, by typing:

q

Remember that core file we created when the program seg faulted in normal (non-debugging) use? That core file contains the information necessary to create the state of the program at the time of improper termination. We can use gdb to access that information by passing the file name as the second parameter to gdb. Now type:

gdb a.out core

You should see something like the following at the end of the introductory display:

```
#0  0x4006af3b in _IO_vfscanf (s=0x401098a0, format=0x8048565 "%d",
    argptr=0xbffffaf0, errp=0x0) at vfscanf.c:963
963     vfscanf.c: No such file or directory.
```

If you look back to the display after you ran gdb in the Run section, you will see that this printout is almost the same. Not only is this the same, but *Backtrace* will work the same. Now type

bt

You should see something similar to the following. Note that is virtually identical to the output when we used *Backtrace* earlier!

```
#0  0x4006af3b in _IO_vfscanf (s=0x401098a0, format=0x8048565 "%d",
    argptr=0xbffffaf0, errp=0x0) at vfscanf.c:963
#1  0x4006c8aa in scanf (format=0x8048565 "%d") at scanf.c:33
#2  0x8048423 in main () at main.c:17
(gdb)
```

Again, this is virtually identical to the output you had when you tested the program within gdb. You can now see that when your program terminates improperly, you can use gdb to easily locate the problem line! Of course, you are not usually going to have a numbered printout of your program at hand. The gdb *List* command eliminates the need for one.

The l (List) Command

You can use this to list parts of your source file(s). For example, typing "*l 52*" will result in display of Line 52 of the current file and the few lines surrounding it (to see more lines, hit the carriage return again). Type now:

l 17

You should see something like:

```
12      in vfscanf.c
(gdb)
```

This wasn't very useful, because we are not interested in vfscanf.c. To change to a different source file, you must precede the line number by the file name and a colon. Now type:

l main.c:17

You should see:

```
12      int main()
13      {
14       int i;
15
16       printf("Enter upper bound:\n");
17       scanf("%d", UpperBound);
18
19       Prime[1] = 1;
20       Prime[2] = 1;
21
```

You can also specify a function name after *list*, in which case the listing will display the lines surrounding the first line of the function. Now type:

l main

You should see the following:

```
8
9       int Prime[50],  /* Prime[i] will be 1 if i is prime, 0 otherwise */
```

```
10       UpperBound; /* check all number up through this one for primeness */
11
12      int main()
13      {
14       int i;
15
16       printf("Enter upper bound:\n");
17       scanf("%d", UpperBound);
(gdb)
```

Now, take a closer look at line #17. Yep, a famous ``C-learner's error''-we forgot the ampersand before UpperBound! The line should have been:
scanf(``%d'',&UpperBound);

So, in the other window, open your favorite text editor and fix line 17 of main.c, and then recompile by typing:

gcc –g main.c prime.c

Note that we do not leave gdb while doing this, since gdb takes a long time to load. After fixing and recompiling main.c, the next time we give gdb the run command 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). Now run the program in gdb window by typing the gdb command:

r

Enter **20** at the prompt again. You should see something like the following:

```
Enter upper bound:
20

Program received signal SIGSEGV, Segmentation fault.
0x80484b1 in CheckPrime (K=3, Prime=0x80496a0) at prime.c:13
13          if (Prime[J] == 1)
(gdb)
```

Now, remember, as mentioned earlier, one of the most common causes of a seg fault is a wildly-erroneous array index. The program tells us that the seg fault occurred on Line #13 of prime.c in the CheckPrime() function. Looking at that line we see an array, Prime, being accessed. Thus we should be highly suspicious of J in this case, and should check what its value is, using gdb's *Print* command.

The p (Print) Command:

This prints out the value of the indicated variable or expression. 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. 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 "*p L*" when you are not in F, you will get an error message like ``No variable L in the present context.'' Take a look at the value of J by typing:

p J

You should see something like (please realize that your value may be different, but will be large):

```
$1 = 592
(gdb)
```

Wow! Remember, I only had set up the array Prime to contain 50 integers, and yet here we are trying to access Prime[592]! 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 CheckPrime function by typing:

l CheckPrime

Then just press the **Enter key** to display the next ten lines of the function. Your display should be be something like:

```
(gdb) l CheckPrime
1       void CheckPrime(int K, int Prime[])
2       {
3         int J;
4
5         /* the plan:  see if J divides K, for all values J which are
6             (a) themselves prime (no need to try J if it is nonprime), and
7             (b) less than or equal to sqrt(K) (if K has a divisor larger
8                 than this square root, it must also have a smaller one,
9                 so no need to check for larger ones) */
10
(gdb)
11        J = 1;
12        while (1)  {
13          if (Prime[J] == 1)
14            if (J % K == 0)  {
15              Prime[K] = 0;
16              return;
17            } /* if */
18          J++;
19        } /* while */
20
(gdb)
```

Look at the comment in Lines #5. We were supposed to be dividing K by J. Looking at our Line #14, we see that we are dividing J by K with our modulo operator! In your text editor change Line #14 to read "if (K % J == 0) {", Before recompiling, we must first tell gdb to relinquish our executable file using the *Kill* command. 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. Now *Kill* the gdb debug and confirm by typing:

k
y

Now, recompile again. Let's *Run* the program again by typing:

r
As before, enter **20** at the prompt. The program output should look like the following:

```
Enter upper bound:
20

Program exited normally.
(gdb)
```

What?! No primes reported up to the number 20? That's not right. Let's use gdb to step through the program.

The b (Breakpoint) Command

The *Breakpoint* command 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. As with the *List* command, if you have more than one source file, precede the line number by the file name and a colon, e.g. "b prime.c: 9". You can also use a function name to specify a breakpoint, meaning the first executable line in the function, e.g., "b CheckPrime". We want to pause at the beginning of main(), and take a look around. So type:

b main

You should see something like:

```
Breakpoint 1 at 0x8048406: file main.c, line 16.
(gdb)
```

So, gdb will pause execution of our program whenever it hits Line 16 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 by typing:

r

We see that, as planned, gdb did stop at the first line of main() (Line 16), and the following is displayed:

.

```
Breakpoint 1, main () at main.c:16
16        printf("Enter upper bound:\n");
(gdb)
```

Now we would like to execute the program one line at a time. The *Next* and *Step* commands permit us to do this.

The n (Next) and s (Step) Commands

Both the *Next* and *Step* commands tell gdb to execute the next line of the program, and then pause again. If that line happens to be a function call, then *Next* and *Step* will give different results. If you use *Step*, then the next pause will be at the first line of the function; if you use *Next*, then the next pause will be at the line <u>following the function call</u> (the function will be 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 *Next*, so that you don't waste a lot of time single-stepping within the function itself. When you use *Step* 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. Now use the *Next* command and type:

n

You will see:

```
Enter upper bound:
17        scanf("%d", &UpperBound);
(gdb)
```

What happened was that gdb executed Line #16 of main.c (the call to printf) as requested, and then paused at the next line, the scanf(). OK, let's execute Line #17, by using the *Next* command again. Since we are executing a scanf, you will have to enter an integer before the program can complete Line #17. Now type:

n
20

Your screen should look like:

```
(gdb) n
```

```
20
19        Prime[1] = 1;
(gdb)
```

As expected, the gdb paused at the next executable line, Line #19. 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 *Print* command, so type:

p UpperBound

Your screen should look like:

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

OK, that's fine. So, let's continue to execute the program one line at a time. Since we are not calling any functions, both the *Next* and *Step* commands will do exactly the same thing. Let's get some experience with the *Step* command, type:

s

As expected we paused at the next executable line, Line #20. The screen should look like:

```
(gdb) s
20        Prime[2] = 1;
(gdb)
```

Use the *Step* command twice more by typing:

s
s

The screen should look like the following:

```
(gdb) s
22        for (i = 3; i <= UpperBound; i += 2)
(gdb) s
23          CheckPrime(i, Prime);
(gdb)
```

Now we have paused at the call of CheckPrime(). Since we think we found all the bugs in CheckPrime(), lets just use *Next* to run it without pausing.

n

The screen should look like the following:

```
(gdb) n
24          if (Prime[i])
(gdb)
```

Let's *Step* once more.

s

The screen should look like the following:

```
(gdb) s
22        for (i = 3; i <= UpperBound; i += 2) {
(gdb)
```

Hey! We didn't execute the printf statement, even though we know that 3 is prime! Let's take a quick look at the Prime array using the *Print* command. Type:

p Prime[3]

The screen should look like the following:

```
(gdb) p Prime[3]
$2 = 0
(gdb)
```

According to our comments at the top of main.c, the Prime array should be set to 1 when a number is prime. Looks like CheckPrime() still has a bug in it. We want to re-run the program and pause at CheckPrime() this time. Set a new breakpoint at the beginning of CheckPrime() by typing:

b CheckPrime

The screen should look like the following:

```
(gdb) b CheckPrime
Breakpoint 2 at 0x80484a6: file prime.c, line 11.
(gdb)
```

To restart the program we must first use the *Kill* command, confirm you wish to kill it and then *Run* it by typing:

k
y
r

The screen should look similar to the following (note that the directory of a.out would be different for you):

```
(gdb) k
Kill the program being debugged? (y or n) y
(gdb) r
Starting program: /home/davis/gdb/temp/a.out

Breakpoint 1, main () at main.c:16
16        printf("Enter upper bound:\n");
(gdb)
```

The c (Continue) Command

Since we are fairly sure there are no bugs at the beginning of the program, we would prefer to not single-step through the lines of code before the next breakpoint. The *Continue* command permits the program to run at full speed until it reaches another breakpoint. Now type:

c
20

The screen should look similar to the following:

```
(gdb) c
Continuing.
Enter upper bound:
20

Breakpoint 2, CheckPrime (K=3, Prime=0x80496c0) at prime.c:11
11        J = 1;
(gdb)
```

Note that we automatically see the values of the parameters passed to CheckPrime(). We can easily confirm that K has the value 3.

The disp (Display) Command

Since we are about to enter a for loop, it would be wise to keep track of the value of loop control variable, J. If we use the *Print* command, we have press *p* after each step. The *Display* commands displays the values of variables automatically at each pause. To watch the J variable type:

disp J

The screen should look like (note that the value of J will be some garbage.):

```
(gdb) disp J
1: J = 1074835916
(gdb)
```

Now use the *Step* command again:

s

You should see:

```
(gdb) s
12      while (1)  {
1: J = 1
(gdb)
```

As expected, J has been initialized to 1. Let's take another three *Steps*:

s
s
s

You should see:

```
(gdb) s
13        if (Prime[J] == 1)
1: J = 1
(gdb) s
14          if (K % J == 0)  {
1: J = 1
(gdb) s
15            Prime[K] = 0;
1: J = 1
(gdb)
```

Now wait a minute! K is three, and according to our specifications in main we assign 0 to Prime[K] when it is not prime! Looking at the "if K % J == 0", we can see that it will always be true if we start J at 1. We need J to start at 2 instead of 1 in our loop. Use your text editor to change Line #11 to "J = 2;".

The undisp (Undisplay) Command

Since we will no longer want to look at value of J repeatedly again, we should use the *Undisplay* command. Each time a variable is displayed it is preceded by it *Display* number and a colon. You use this number to turn off its display. The variable *J* is preceded by 1:, so its *Display* number is 1. To *Undisplay* it by type:

undisp 1

Nothing will appear on the screen, except the next "(gdb)" prompt. You will need to *Kill* the debug session before recompiling by typing:

k
y

Now recompile.

The d (Delete) Command

Since we are sure the beginning of the program contains no bugs, there is no need for the breakpoint at its start. We can use the *Delete* command to remove breakpoints. We follow the command with the breakpoint number(s) that we wish to remove. If we do not supply any numbers, then all breakpoints are deleted. As we noted at the time, our breakpoint at the beginning of main() was #1, so type:

d 1

Nothing will appear on the screen, except the next "(gdb)" prompt. Now *Run* the program and enter 20 again by typing:

r
20

The screen should look similar toe the following:

```
(gdb) r
`/home/davis/gdb/temp/a.out' has changed; re-reading symbols.
Starting program: /home/davis/gdb/temp/a.out
Enter upper bound:
20

Breakpoint 2, CheckPrime (K=3, Prime=0x80496c0) at prime.c:11
11        J = 2;
(gdb)
```

Now let's just try *Continue* by typing:

c

You should now see:

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
```

```
0x80484c1 in CheckPrime (K=3, Prime=0x80496c0) at prime.c:13
13          if (Prime[J] == 1)
(gdb)
```

Not another Segmentation fault! Since a line that includes an array is again suspect we should check the value of J by using the *Print* command:

p J

The screen should look like:

```
(gdb) p J
$1 = 592
(gdb)
```

The value of J is huge, (yours may be different). Our array, Prime, is designed to hold only fifty ints. If we read the comments on Lines #5 to #9 of prime.c, then we see that J should stop when it reaches the square root of K. Yet you can see in Lines #12 to #17 that we never made this check, so J just kept growing and growing, eventually reaching the value 592 which triggered the seg fault. In your text editor, delete the "}" on Line #19, delete the "J++" on Line #18, delete the "J = 2" on Line #11, and change the while statement on Line #12 to read "*for (J = 2; J * J <= K; J++)*". When you are done, that part of your code should look like:

```
  for(J = 2; J * J <= K; J++)
    if (Prime[J] == 1)
      if (K % J == 0)  {
        Prime[K] = 0;
        return;
      } /* if */

  /* if we get here, then there were no divisors of K, so it is  prime */
```

Before you re-compile your program, you will need to *Kill* the debug session by typing:

k
y

Since we think we've pretty well taken care of all the bugs, let's get rid of all the breakpoints using *Delete* by typing:

d
y

The screen should look like the following:

```
(gdb) d
Delete all breakpoints? (y or n) y
(gdb)
```

Now let's run at full speed using *Run* by typing:

r
20

If all goes well, your screen should look like the following!!!

```
`/home/davis/gdb/temp/a.out' has changed; re-reading symbols.
Starting program: /home/davis/gdb/temp/a.out
Enter upper bound:
```

```
20
3 is a prime
5 is a prime
7 is a prime
11 is a prime
13 is a prime
17 is a prime
19 is a prime

Program exited normally.
(gdb)
```

Looks like we are done! Time to *Quit* gdb by typing:

q

If you are doing this tutorial for a class, type **exit** in the gdb window. This will terminate the *script* program, and you should see a message "`Script done, file is typescript`". The file named *typescript* in the current directory. You will handin main.c, prime.c, and typescript to prove you did this tutorial.

Beyond Basic gdb

Advanced gdb Commands

As you may have noticed when you typed the *Help* command, there are many gdb commands that we did not cover here. Of particular note are *Set*, *Printf*, and *Define*. Professor Matloff describes these in detail in the source document for this tutorial: http://heather.cs.ucdavis.edu/~matloff/debug.html

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.

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.[1] 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 that 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.

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 most platforms is Eclipse. You can find information about Eclipse at http://heather.cs.ucdavis.edu/~matloff/eclipse.html 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.

What Does Professor Matloff Use?

Mostly Professor Matloff uses *gdb*, accessed from *ddd*, along with *vim* for his editor. He occasionally uses Eclipse.

main.c

```
/* 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.   It is riddled with errors! */

#include <stdio.h>
void CheckPrime(int K, int Prime[]); /* prototype for function in prime.c */

int Prime[50],  /* Prime[i] will be 1 if i is prime, 0 otherwise */
UpperBound; /* check all number up through this one for primeness */

int main()
{
  int i,

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

  Prime[1] = 1;
  Prime[2] = 1;

  for (i = 3; i <= UpperBound; i += 2) {
    CheckPrime(i, Prime);
    if (Prime[i])
      printf("%d is a prime\n", i);
  }
  return 0;
} /* main() */
```

prime.c

```
CheckPrime(int K, int Prime[])
{
  int J;

  /* the plan:  see if J divides K, for all values J which are
  themselves prime (no need to try J if it is nonprime), and
  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 = 1;
  while (1)  {
    if (Prime[J] == 1)
      if (J % K == 0)  {
        Prime[K] = 0;
        return;
      } /* if */
    J++;
  } /* while */

  /* if we get here, then there were no divisors of K, so it is  prime */
  Prime[K] = 1;
} /* CheckPrime() */
```