

# Using the GDB Debugger

## 1 Overview

The webpage for the GDB debugger (<http://www.gnu.org/software/gdb/>) provides this introduction:

“GDB, the GNU Project debugger, allows you to see what is going on ‘inside’ another program while it executes — or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

The program being debugged can be written in C, C++ (and many other languages). Those programs might be executing on the same machine as GDB (native) or on another machine (remote). GDB can run on most popular UNIX and Microsoft Windows variants.”

## 2 Example

We’ll use a contrived example in C++ based on the C code from the 2002 version of “Guide to Faster, Less Frustrating Debugging” by Norman Matloff (you can find this with a Google search). The source file is `check_primes.cpp`. The program, when working, will output a list of all primes less than or equal to an upper bound supplied by the user.

1. Compile and link the program using `make_check_primes`. The makefile is set to run without any of our usual warning switches; you should see this:

```
g++ -c check_primes.cpp -o check_primes.o
g++ -o check_primes check_primes.o
```

If we used all of the warning options from our standard Makefile, we would catch many of the errors in `check_primes.cpp` at compile time. Since we want to illustrate gdb, we’ll compile without those options (but you should **ALWAYS** use the warnings; `-Wall` will take care of most of the problems).

2. Try running the program `check_primes`. When it asks for an upper bound, enter “20” (without the quotes!). The program will exit with a “Segmentation fault”. This error means that the program tried to dereference a pointer containing a bad value. (What? That means that the pointer pointed to a place it shouldn’t be pointing. Examples below!) Note that at this point you have *no idea* where the error occurred in the program. GDB to the rescue!
3. Start up GDB with the name of the executable on the command line:
 

```
gdb check_primes
```

 This will result in several lines of information and then a gdb prompt:
 

```
(gdb)
```

 at which you can type commands.
4. We’ll start with the “run” command (which can be abbreviated as “r”). You’ll get an informational line about what program is being started and about “(no debugging symbols found)” and then the “enter upper bound” prompt should appear eventually. Enter “20” as before. The program fails again with a “Segmentation Fault” but now there is more information. However, the information might not appear to be very helpful! In particular, the message seems to say the error is in some function that is not one of the ones we have defined. It must have been called by one of the C++ library functions that our program has called.

To see which one, you can in most cases use the gdb `bt` (“backtrace”) command:

```
(gdb) bt
```

This should tell you the function it was called from. It would also tell you what line, but we’ve forgotten to include the debugging symbols (which gdb warned us about)! “Quit” gdb with `(gdb) q`.

*[In general you can leave gdb running in one terminal window while running the editor from another by first giving the command:*

```
(gdb) kill
```

*and, after recompiling `check_primes.cpp`, giving the run (`r`) command again.]*

5. Edit `make_check_primes` and locate the instructions for compiling `check_primes.o`. Replace `NOCFLAGS` by `CFLAGS` and check the definition of `CFLAGS` earlier in the file. The “-g” switch is needed to provide gdb with information such as line numbers. Always use it when developing a code. We use “-O0” rather than some higher optimization because we want to turn off all optimization (which can interfere with our debugging). We’ll put it back when everything is working.

Now recompile, run gdb with `check_primes`, enter 20, and `bt` again. This time you should be told at what line number in what function the segmentation fault occurred and in what source (.cpp) file that function is located. (The original call is the last one listed; start there.)

6. To help identify the problem with a particular line, use the gdb `l` (“list”) command. For example, if the problem is identified as being at `check_primes.cpp:20`, then you

want to list near line 20:

```
(gdb) l check_primes.cpp:20      or simply      (gdb) l 20
```

List the appropriate lines in this case based on what gdb tells you. Can you identify the error (which is a very common mistake with C input functions!)? This is an error that `-Wall` would have detected. “Quit” gdb with `(gdb) q` (or “kill” the program), then fix the error in an editor and recompile the program. (If you don’t know how to fix the C error, replace it with a C++ input statement using `cin`.)

7. Ok, run in gdb again, enter “20” as the upper bound again and ... “Segmentation fault” again! At this point you might normally insert `cout` statements to identify the values of the variables in the suspect line (or nearby). We can do that *within* gdb with the `p` (“print”) command. To print the value of `N`, for example:

```
(gdb) p N
```

Look at the nearby lines (using “list”) to see where we are in the code. Try printing `true_ptr` and `*true_ptr`. Recall that a pointer like `true_ptr` stores an address and `*true_ptr` *dereferences* that address; that is, it tells you what is stored there. So you might expect that printing `*true_ptr` would give the value 1. The problem is that we have tried to *dereference an uninitialized pointer*. That is, `true_ptr` never got assigned an address! Here are two ways to fix it:

```
int true_value = 1; // declare a variable first
int *true_ptr = 0; // Note: should always initialize to "null pointer"
true_ptr = &true_value; // set true_ptr to address of true_value
```

or “dynamically allocate” `true_ptr` using the `new` command:

```
int *true_ptr = new int; // allocate an address using "new"
*true_ptr = 1;           // set value pointed at
```

Pick one of these to fix the code.

8. Let’s see how to step through the program and see what is happening line by line. To do so, we first set a “breakpoint,” which is a place where gdb will suspend execution of the program, so we can check the values of variables and then continue line by line. We can set breakpoints at the start of functions (e.g., “b main” or “b CheckPrime”) or at specific line numbers (e.g., “b check\_point.c:71”). In this case, we’ll start from the beginning. Kill the program as described above, run again, and at the first prompt set a breakpoint and then run:

```
(gdb) b main
```

```
(gdb) r
```

which should take you to Breakpoint 1 (line 40 or so). To proceed, we can use either of two types of stepping commands: `s` (“step”) or `n` (“next”). Each of these commands advance gdb one executable line in the program. The difference is that `n` will not step into functions called by the program but will just execute the function and then continue. This avoids wasting time in functions you believe are working correctly. Here we’ll use `s` repeatedly:

(gdb) s

(gdb) s

and so on. Each time you give an `s` command, gdb will tell you where it is. Step along, entering “20” when requested, and continue to where `CheckPrime` is called, and keep going *into* the `CheckPrime` function. What is `K` just after you enter the function? (Use “print” to check.) Is it correct? What is `J` at this point? What does that mean?

9. Now “continue” running by typing `c`. Guess what: segmentation fault. (Who wrote this mess?) You should be told there is a problem in `check_primes.cpp` with a line number that includes an array reference. Note that one of the most common sources of segmentation faults is an array index that is “out of bounds.” That is, the value of the index is negative or larger than the declared size of the array. Figure out what is out of bounds (use “print”). You should find that `J` has gotten very large; how did that happen? Check the plan of the code in the comments to the `CheckPrime` function. Is the plan (part (b) in particular) implemented? If not, change the code so that it is implemented correctly [the quickest fix involves changing the expression in `()`’s for the `while` loop]. Quit gdb, edit the code, recompile, and run again.

10. Ok, now the program executes and exits but with no output! Now we’re going to step through the code again to identify why nothing is output. Kill the program and restart it, then set a breakpoint in `main`. This time we’ll use `n` to step through since we think the `CheckPrime` function is working. We’ll also keep track of the value of `N` by using the `disp` (“display”) command. So, after restarting gdb:

(gdb) b main

(gdb) r

(gdb) disp N

(gdb) n

and we can keep entering `n` or a return to step through lines. Note that before `N` is initialized in the `for` loop, it may have an interesting value! Continue entering `n`’s. After entering “20” yet again, we see that `CheckPrime` is called and then we are back to the start of the `for` loop. (We don’t go into `CheckPrime` because we’re using `n` instead of `s`.) Is it correct that we should be back to the loop? Aren’t we supposed to be printing out results as we go? Kill the program in gdb and fix the problem!

11. One more time: recompile, go back to gdb, and run. Hopefully it will execute correctly and our work here is done.