Introduction to gdb

Dr. Yihsiang Liow (August 5, 2023)

Contents

1 Introduction

gdb (GNU Debugger) is a power debugger for the linux platform. A debugger is a tool that is frequently helpful in finding bugs. Here's the gdb entry in wikipedia:

The GNU Debugger, usually called just GDB and named gdb as an executable file, is the standard debugger for the GNU operating system. However, its use is not strictly limited to the GNU operating system; it is a portable debugger that runs on many Unix-like systems and works for many programming languages, including Ada, C, C++, Objective-C, Free Pascal, Fortran, Java and partially others.

– *Wikipedia*

A debugger might be helpful in locating a bug. Of course using print statements help too. But a debugger might save you some time.

I will use `$` to denote the bash shell prompt. For instance if I say execute the `ls` command at the prompt I would write

```
$ ls
```

To make sure you have gdb, run `gdb` in your bash shell and then quit by doing Ctrl-D. If gdb is not found, as root, you should install gdb by executing

```
dnf -y install gdb
```

In the next few sections on using gdb, gdb might give you a warning such as

```
Missing separate debuginfos, use: dnf debuginfo-install libasan-9.3.1-2 [... etc. ...]
```

That means gdb needs the libraries `libasan-9.3.1-2.fc31.x86_64`, etc. In that case, as root, execute the above command gdb gave you:

```
dnf debuginfo-install libasan-9.3.1-2 [... etc. ...]
```

2 Starting and Quitting

First write `helloworld.cpp`:

```
#include <iostream>

int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

Now compile the above with the `-g` option:

```
[student@localhost test] g++ helloworld.cpp -g -o helloworld
```

The executable binary file is `helloworld`. Now run gdb on helloworld:

```
[student@localhost test] gdb helloworld
```

You will see the gdb prompt where you can execute commands:

```
(gdb)
```

To quit gdb you type `q` at the gdb prompt.

Exercise 2.1. Quit gdb and then run it with `helloworld` again.

Note that if you have to run your program with command line arguments with input data from a file like this:

```
[student@localhost test] helloworld < input.txt
```

then you should run gdb like this:

```
[student@localhost test] gdb helloworld < input.txt
```

3 Listing

You can view your source in gdb. Here's one way of doing it. Type

```
(gdb) list
```

at the gdb prompt Try it now. You see that gdb will list your program. You can also do

```
(gdb) l
```

(that's the letter l and not the number 1!)

If you do `list` again, gdb will tell you that you at the end of your source file. So `list` list from the last line of your source file. By default gdb list 10 lines.

Another way is to specify a line number like this:

```
(gdb) l 3
```

gdb will list the source around line 3 of your program. You can specify a starting and ending line number like this:

```
(gdb) l 3, 5
```

That will list lines 3, 4, and 5. To list only line 3, you can do:

```
(gdb) l 3, 3
```

Here are some other options:

```
l 3,        -- lists starting at line 3
l ,3        -- lists up to line 3
l +         -- lists after this point
l -         -- lists before this point
```

Now modify your program:

```
#include <iostream>

void foo()
{
    std::cout << "foo" << std::endl;
}

int bar()
{
    std::cout << "bar" << std::endl;
    return 42;
}
```

```
double baz()
{
    std::cout << "baz" << std::endl;
    return 3.14;
}

int main()
{
    std::cout << "hello world" << std::endl;
    foo();
    bar();
    baz();
    return 0;
}
```

Again compile your program with the -g option. Run gdb on the executable. And execute `l` several times. Notice that each time you run `l`, you get about 10 lines.

This time try to list your source around the function baz like this:

```
(gdb) l baz
```

Exercise 3.1. Can gdb list source near a class?

If you have more than one source file and one of them is named `ABC.cpp` then you can specify the source file name in the list command like this:

```
(gdb) l ABC.cpp:3
```

This will list ABC.cpp starting at line 3. Or you can do this:

```
(gdb) l ABC.cpp:f
```

This will list `ABC.cpp` starting around function `f`.

4 Running a Program

Recall our `helloworld.cpp` looks like this:

```
#include <iostream>

void foo()
{
    std::cout << "foo" << std::endl;
}

int bar()
{
    std::cout << "bar" << std::endl;
    return 42;
}

int baz()
{
    std::cout << "baz" << std::endl;
    return 3;
}

int main()
{
    std::cout << "hello world" << std::endl;
    foo();
    bar();
    baz();
    return 0;
}
```

I'm assuming you have gdb running the executable of the above source. To run your executable within gdb type this at the gdb prompt:

```
(gdb) r
```

No surprises here. (gdb will also show you some extra output including loading of libraries.)

If you need to run your program with some input file, say input.txt, you do thisin gdb:

```
(gdb) r < input.txt
```

5 Breakpoints

Now let's set a breakpoint. (You'll see what it does in a minute.) The function call

```
foo();
```

is at line 23. Verify this using the l command. Let's set a breakpoint at line 23. At the gdb prompt type this:

```
(gdb) b 23
```

gdb will tell you that you have a breakpoint (breakpoint number 1) at line 23. Now run the program with the r command. The output looks like this:

```
hello world

Breakpoint 1, main () at helloworld.cpp:23
23          foo();
```

Now continue the execution of your program by doing this at the gdb prompt:

```
(gdb) c
```

and you get this output:

```
Continuing.
foo
bar
baz
Program exited normally.
```

So a breakpoint is just a place where you want the program execution to stop temporarily.

Exercise 5.1. Set another breakpoint at the only statement within the foo() function:

```
void foo()
{
    std::cout << "foo" << std::endl; // set a breakpoint here
}
```

Run your program again, continuing until the program terminates. Note that you have to continue twice.

You can also set a breakpoint with a function name. Try this:

```
(gdb) b bar
```

This will put a breakpoint at the first statement of the function `bar()`. Run your program again.

Exercise 5.2. Write a program containing a class with a method. Verify that you can insert a breakpoint at the method.

Now for disabling a breakpoint. You can do that using:

```
(gdb) disable 1
```

or just

```
(gdb) d 1
```

The "1" here refers to breakpoint 1 and not some line number. Run your program again. You can enable a breakpoint after disabling it.

```
(gdb) enable 1
```

Run your program again. As you can see, disabling a breakpoint does not remove it. It's just not active. If you want to remove a breakpoint by line number you do this:

```
(gdb) clear 23
```

You can list all breakpoints by doing

```
info b
```

If you are working with multiple files, you include the name of the cpp file. For instance

```
(gdb) b ABC.cpp:10
```

sets a breakpoint at line 10 of `ABC.cpp`.

6 Viewing Program State

Why do we want to set breakpoints? Because we want to view the state of the program which is just a fancy way of saying we want to see the values of the variables in our program at the point when the program stops. This is useful in a debugging process: You set breakpoints just before the point where you suspect something is wrong, run the program up to that point, and view the values of the variables to see if values matches your expectation.

Modify your helloworld.cpp as follows:

```
#include <iostream>

void foo()
{
    std::cout << "foo" << std::endl;
}

int bar()
{
    std::cout << "bar" << std::endl;
    return 42;
}

int baz()
{
    std::cout << "baz" << std::endl;
    return 3;
}

int main()
{
    std::cout << "hello world" << std::endl;
    int x = 1;
    foo();
    x = x + 2;
    x = bar();
    x = baz();
    return 0;
}
```

Recompile your program and run gdb on the executable. First set a breakpoint before the function call

```
foo()
```

Next execute the program. When gdb stops at the breakpoint, display the value of variable x using the print command:

```
p x
```

You will see that the value of x is 1:

```
$1 = 1
```

Of course you can also print the value inside an array. For instance if `a` is an array (or a pointer), you can do

```
p a[1]
```

You can also print the whole array. To print the first 5 values of array `a`, you execute

```
p *a@5
```

You can also print the values of an object.

There are times when you know that something is wrong with the value of a single variable, say `x`, and you want to know the behavior of this variable. You can of course execute "`p x`" again and again. But there's an easier way to do this. You can get gdb to display the value of `x` at every breakpoint. You do that with this command:

```
disp x
```

Exercise 6.1. Suppose now that you have fixed the problem with `x` and you just want to watch the changing values of `y`. What is the command to stop displaying `x` at every breakpoint?

7 Next and Step Command

Frequently you only know roughly the location of a bug. What you want to do is to pause the execution of your program before the place where a bug is lurking and then execute the program one statement at a time. You can of course set a breakpoint for each statement near the place where the bug might be. A better way to do this is the set a single breakpoint and then get gdb to pause at the beginning of each statement.

Here's our program again:

```
#include <iostream>

void foo()
{
    std::cout << "foo" << std::endl;
}

int bar()
{
    std::cout << "bar" << std::endl;
    return 42;
}

int baz()
{
    std::cout << "baz" << std::endl;
    return 3;
}

int main()
{
    std::cout << "hello world" << std::endl;
    int x = 1;
    foo();
    x = x + 2;
    x = bar();
    x = baz();
    return 0;
}
```

First place a breakpoint at the function call to foo(). Next, run the program. When you reach the breakpoint, run the next command:

```
(gdb) n
```

several times until you reach the end of your program. You see that the program executes, pausing at each statement in your current function, i.e. `main()`. The important thing is that there is no pause within the functions that `main()` calls. For instance there is no pause inside `foo()`.

The other command, `step`, will in fact pause at every single statement. Try it out. The problem is that the print statement in `foo()` called many other functions. It will take a long time to execute your program. So it you believe that your code contains the bug and not the function(s) that you use, you might want to use the `next` command instead of `step`.

You can speed things up by specifying a number after the `next` and `step` command. For instance `step 5` will execute 5 steps before pausing.

8 Setting the Value of a Variable

You can set the value of a variable too. Here's our program again:

```
#include <iostream>

void foo()
{
    std::cout << "foo" << std::endl;
}

int bar()
{
    std::cout << "bar" << std::endl;
    return 42;
}

int baz()
{
    std::cout << "baz" << std::endl;
    return 3;
}

int main()
{
    std::cout << "hello world" << std::endl;
    int x = 1;
    foo();
    x = x + 2;
    x = bar();
    x = baz();
    return 0;
}
```

Set a breakpoint at the function call to `foo()`. Run the program. At the breakpoint set the value of `x` to 1980:

```
(gdb) set variable x = 1980
```

set display of x to on:

```
(gdb) disp x
```

and issue the next command:

```
(gdb) n
```

twice and you will see that `x = x + 2` will give you `1982` for the value of `x`.

9 Backtrace

A program execution usually weaves through lots of function calls. The backtrace command tells you where the program execution has gone through.

Modify our program as follows:

```
#include <iostream>

void foo2(int x)
{
    std::cout << "foo2" << std::endl;
    std::cout << 1 / x << std::endl;
}

void foo(int x)
{
    std::cout << "foo" << std::endl;
    --x;
    foo2(x);
}

int bar()
{
    std::cout << "bar" << std::endl;
    return 42;
}

double baz()
{
    std::cout << "baz" << std::endl;
    return 3.14;
}

int main()
{
    std::cout << "hello world" << std::endl;
    int x = 1;
    foo(x);
    x = x + 2;
    x = bar();
    x = baz();
```

```
    return 0;
}
```

First compile and run your program in your bash shell. You will get a program error. You do get an error message which might help. Now let's get gdb to run it and see what happens.

Run your program in gdb. Look at the error message.

```
Program received signal SIGFPE, Arithmetic exception.
0x00000000004007f4 in foo2 (x=0) at main.cpp:6
6            std::cout << 1 / x << std::endl;
```

Much better right?

Also, execute the following gdb command:

```
(gdb) bt
```

and you will see the function call stack, i.e., the functions starting at `main()` up to the function where the error occurs:

```
#0  0x00000000004007f4 in foo2 (x=0) at main.cpp:6
#1  0x0000000000400848 in foo (x=0) at main.cpp:13
#2  0x00000000004008d1 in main () at main.cpp:32
```

(Each function uses a certain amount of memory – this is called a function frame. The function frame are organized as a stack.)

The error message in gdb and the call stack will help you debug your program.

(The gdb command `where` and `info stack` has the same effect as `bt`.)

In case the stack is huge, you can do

```
(gdb) bt 2
```

for the 2 frames closest to the current point of execution. Doing

```
(gdb) bt -2
```

would be the opposite – those frames further from the current point of execution.

At this point, in gdb, you can list the backtrace one at a time going backward by doing

```
(gdb) up
```

There's also the `down` command.

You can run your program again. gdb will ask you if you want to start run the beginning. If you say yes, gdb will run it again. If your program halts half way, you can kill it by doing

```
(gdb) kill
```

As an execise, run gdb on the following program:

```
#include <iostream>

void foo2(int x)
{
    std::cout << "foo2" << std::endl;
    int a[10];
    std::cout << a[x] << '\n';
}

void foo(int x)
{
    std::cout << "foo" << std::endl;
    x *= 1000;
    foo2(x);
}

int bar()
{
    std::cout << "bar" << std::endl;
    return 42;
}

double baz()
{
    std::cout << "baz" << std::endl;
    return 3.14;
}

int main()
{
    std::cout << "hello world" << std::endl;
    int x = 1000;
    foo(x * 5);
    x = x + 2;
    x = bar();
    x = baz();
    return 0;
}
```

10 Conditional Breakpoints

You can set a breakpoint based on a condition. Here's the program to try out:

```
#include <iostream>

int main()
{
    for (int i = 0; i < 100; i++)
    {
        std::cout << i << std::endl;
    }
    std::cout << "end" << std::endl;
    return 0;
}
```

Compile it with the `-g` option and run gdb with the executable. At the gdb prompt, set a conditional breakpoint at line 7 when `i` is 95:

```
(gdb) b 7 if i == 95
```

Now run the program.

The above illustrate how to get to a point in the execution of the program quickly when that point is dependent on a variable and not the line number of the source.

Exercise 10.1. Can you use `<` instead of `==`? What else should you try?

11 Debugging a Simple Arithmetic Error

Modify the program:

```
#include <iostream>

...

int boo(int x, int y, int z)
{
    return x / (y - z);
}

int main()
{
    std::cout << "hello world" << std::endl;
    int x = 1;
    double y = 3.14;
    foo();
    x = x + 2;
    x = bar();
    x = baz();
    x = boo(1, 1, 1);
    return 0;
}
```

Compile it and run the program in gdb. Here's the message:

```
(gdb) r
Starting program: /home/student/gdb/helloworld
hello world
foo
bar
baz

Program received signal SIGFPE, Arithmetic exception.
0x080486bd in boo (x=1, y=1, z=1) at helloworld.cpp:22
22            return x / (y - z);
```

gdb tells you where the error occurs and also the parameter inputs for the function where the error occurs. If you run this in your bash shell this is what you get instead:

```
$ ./helloworld
hello world
foo
bar
baz
Floating point exception (core dumped)
```

which is not as helpful as the message in gdb.

12 Odds and Ends

You can read more about the other commands using the gdb help command: At the gdb prompt do this:

```
(gdb) h
```

Other resources include the man pages and the web.

Exercise 12.1. You run gdb without specifying the program file. What command do you use to load it?

Here's a summary of the commands used in this tutorial.

```
r             run from beginning to the end or to the first breakpoint
b             set breakpoint by line number or function or method
disable       disable a breakpoint (breakpoint is not removed)
enable        enable a breakpoint that was disabled
c             run to the next breakpoint
n             run to beginning of next statement in current function block
s             run to the beginning of next statement
disp          set display of variable whenever a breakpoint is reached
p             print the value of a variable
set variable  set the value of.a variable
bt            print backtrace
h             help
w             sets a watchpoint
```

When I run a opengl program in gdb I get:

```
(gdb) run
Starting program: /mnt/hgfs/yliow/Documents/work/cc/courses/ciss380-graphics/n/opengl
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0xb7fc32f6 in glutInit () from /lib/libglut.so.3
Missing separate debuginfos, use: debuginfo-install freeglut-2.8.0-7.fc18.i686 glibc-
(gdb)
```

In this case, focusing on

```
Missing separate debuginfos, use: debuginfo-install freeglut-2.8.0-7.fc18.i686
```

you login as root and execute

```
debuginfo-install freeglut
```

http://stackoverflow.com/questions/10389988/missing-separate-debuginfos-use-debuginfo