CS354 gdb Tutorial

Written by Chris Feilbach <crf@cs.wisc.edu>

Purpose

This tutorial aims to show you the basics of using gdb to debug C programs. gdb is the GNU debugger, and is provided on systems that have gcc to aid with debugging C programs. This debugger is an incredibly powerful tool and this tutorial will only cover the basics. All of the commands and techniques covered here should be enough to adequately debug all CS354 programs.

Two programs are provided for you to debug. For those of you at the session, it will be structured like a lab – you read through the directions and follow them step by step. I'll be around to answer questions for an hour or two.

There is no specific assembly language example, but useful assembly language commands are provided for your reference for assignment 2, and the methodology for debugging C programs is not very different from assembly language ones.

More information on gdb can be found in the gdb manual here:
https://sourceware.org/gdb/current/onlinedocs/gdb/

What Isn't Covered Here

There are a few things that aren't covered by this document and won't be helped with at the review session:

- Use of basic linux commands, such as how to create a directory, change directories, or how to copy files.
- Using the gcc compiler to compiler C programs.
- Learning how to use a text editor.

At this point in your CS354 career these should be skills you are familiar with.

Getting Started

Start by obtaining the files from http://pages.cs.wisc.edu/~cs354-2/handouts/gdb. Copy these files into a private directory.

Compiling Files for Use with gdb

gdb requires an additional compiler flag to generate something called debug symbols and the removal of the –O flag (optimized code makes it harder to debug). When the compiler generates the machine code that your program executes, it generally discards things like variables, because those don't matter to a computer. However, in order for us to debug properly, we need that information. We add a –g to the end of our gcc command line to do this. Here is an example:

```
gcc –o prog1 prog1.c –m32 –Wall  –g
```

Before Using gdb – Can You Locate the Bugs?

While debugging tools are very handy, it is a valuable skill to be able to look at code and determine whether or not there are mistakes. You should take some time to look at prog1.c and prog2.c and note anything that looks incorrect. **Don't fix these bugs before compiling – even if you see them it is still good practice to use the debugger to verify they are actually bugs.** When you note where things look wrong, keep track of both the name of the file you're examining, and also the line number where the bug occurs. This information will be needed when you start to use gdb.

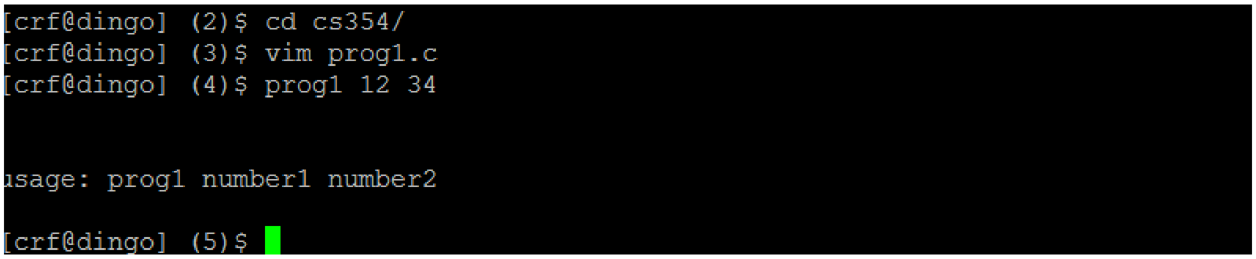
Another useful debugging technique is to compile the programs and run them. Do they behave like you expect them to? If not, what specifically are they doing wrong? By examining what the current program does, you can make guesses as to what might be wrong, and that will limit your time working with the debugger. Don't just stop at the first sign of failure, either! Most bugs in prog1.c can be found by running the program with varying inputs, although some of the bugs are harder to pick up than others.

Another helpful place to look is the warnings that gcc produces, if any. Warnings exist to let you know that while the compiler generated the code, there were things about the code that it didn't like. For example, a warning about an unused variable in code could mean that

Fixing a Bug in prog1.c

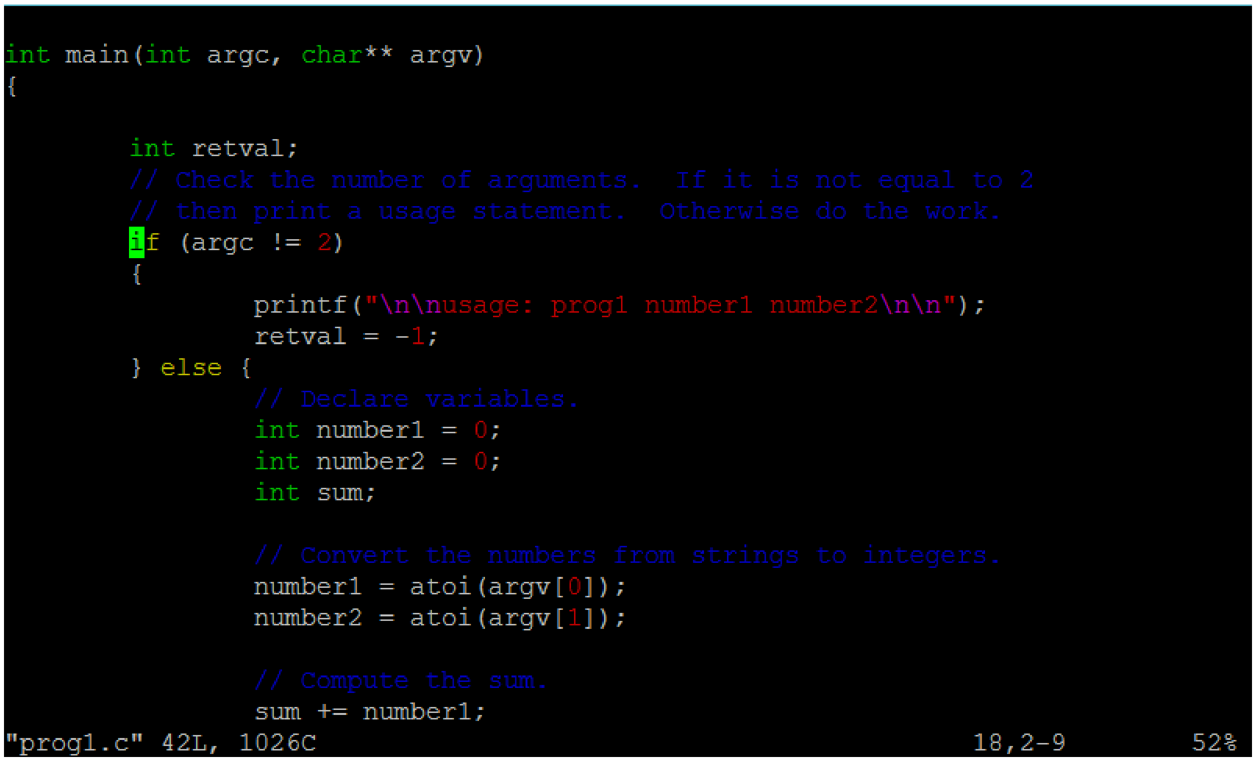
prog1.c is a program that takes two numbers from the command line, and then adds them together and prints the result out to standard output (stdout).

At this point you should have reviewed the code in prog1.c. Let's tackle the first bug that I injected, which can be observed by trying to run prog1 with two numbers after it. However, that doesn't work:



```
[crf@dingo] (2)$ cd cs354/
[crf@dingo] (3)$ vim prog1.c
[crf@dingo] (4)$ prog1 12 34


usage: prog1 number1 number2

[crf@dingo] (5)$
```

So let's look at the code. I found that the usage statement is printed out when the argument argc is not equal to 2. The program is not working, and we can use gdb to find out why.



```
int main(int argc, char** argv)
{

        int retval;
        // Check the number of arguments.  If it is not equal to 2
        // then print a usage statement.  Otherwise do the work.
        if (argc != 2)
        {
                printf("\n\nusage: prog1 number1 number2\n\n");
                retval = -1;
        } else {
                // Declare variables.
                int number1 = 0;
                int number2 = 0;
                int sum;

                // Convert the numbers from strings to integers.
                number1 = atoi(argv[0]);
                number2 = atoi(argv[1]);

                // Compute the sum.
                sum += number1;
"prog1.c" 42L, 1026C                                          18,2-9         52%
```

The first technique that I will go over is how to start gdb and examine the contents of a variable or argument. This is a very useful technique that can be used to solve a large amount of problems. While looking through the program I noticed that the if statement starts at line 18. I am going to make note of this because gcc will need it later on.

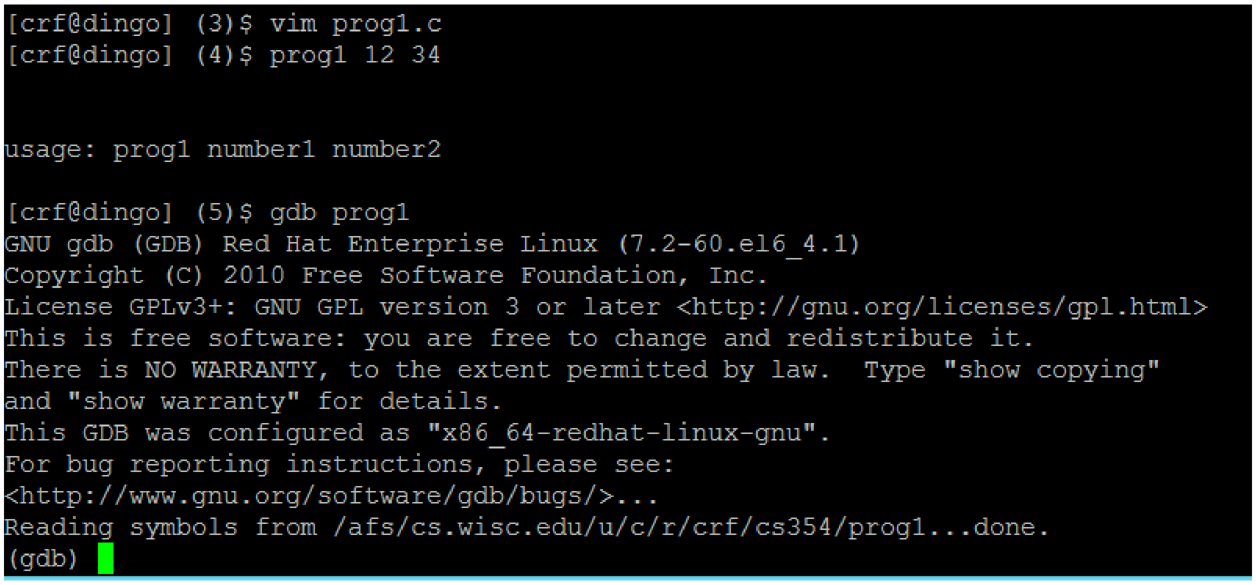
Starting Up gdb

We can run gdb by typing the following in the shell:

gdb prog1

gdb will now load and will be ready to debug the binary prog1.

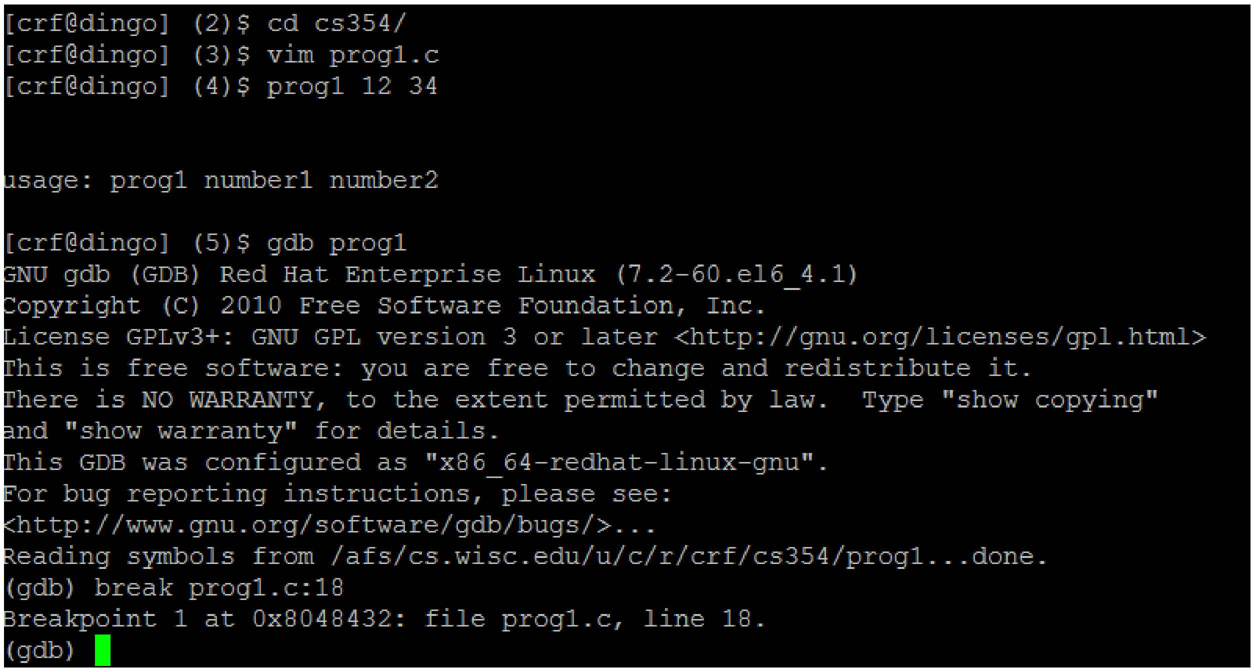
When I debug, I like to open up a second terminal so I can examine the code I'm debugging in vim.


```
[crf@dingo] (3)$ vim prog1.c
[crf@dingo] (4)$ prog1 12 34


usage: prog1 number1 number2

[crf@dingo] (5)$ gdb prog1
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1...done.
(gdb) 
```

Setting a Breakpoint and Running prog1 in gdb

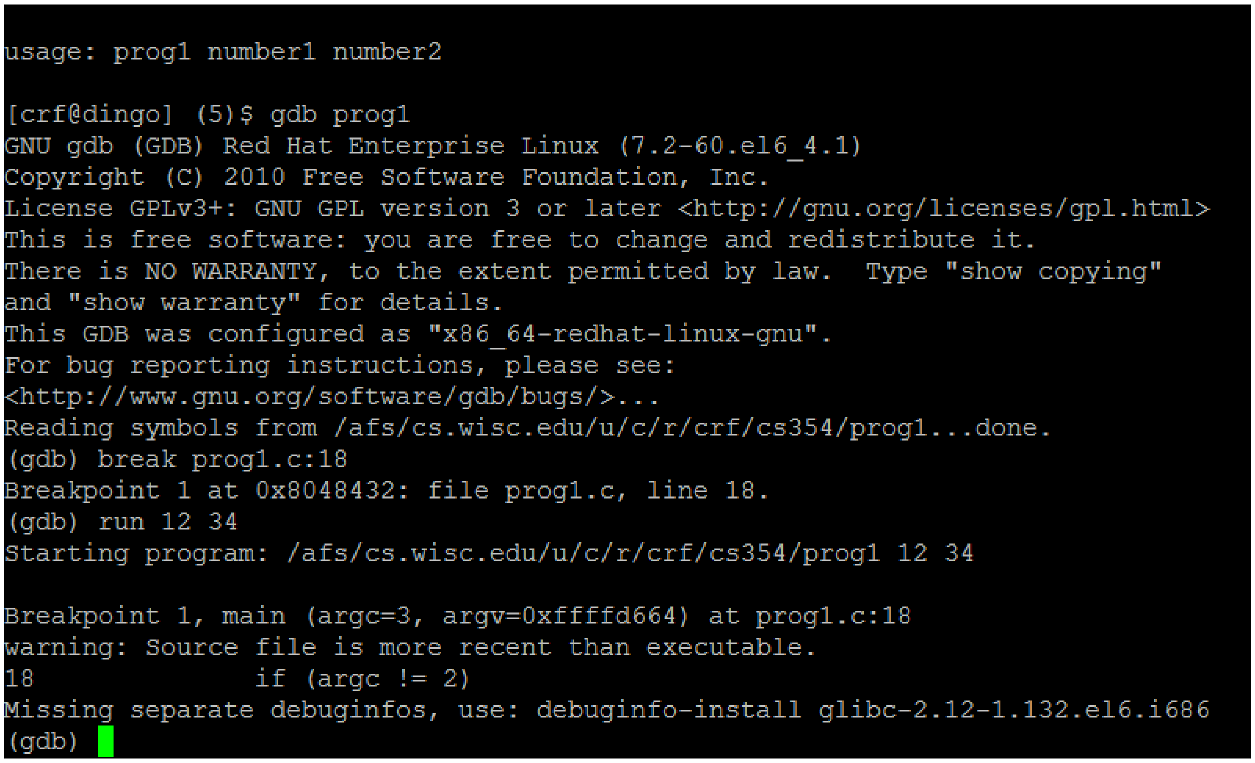
To get gdb to do something for us we must give it commands telling it what to do. We are going to set a breakpoint and then run the program. The program will run until gdb hits the breakpoint, which in our case will be line 18 (where the if statement starts). We tell gdb about the breakpoint by specifying the name of the c file followed by a colon, followed by the line to break on. We enter breakpoint using the break command and then hit enter:

break prog1.c:18


```
[crf@dingo] (2)$ cd cs354/
[crf@dingo] (3)$ vim prog1.c
[crf@dingo] (4)$ prog1 12 34


usage: prog1 number1 number2

[crf@dingo] (5)$ gdb prog1
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1...done.
(gdb) break prog1.c:18
Breakpoint 1 at 0x8048432: file prog1.c, line 18.
(gdb) 
```

gdb let us know that the breakpoint was set. Let’s run the program! We can specify that we want the program to run by giving gdb the run command. If we want to specify command line arguments, we can append them to the run command, just like we did when we ran prog1. Here’s the gdb command:

run 12 34

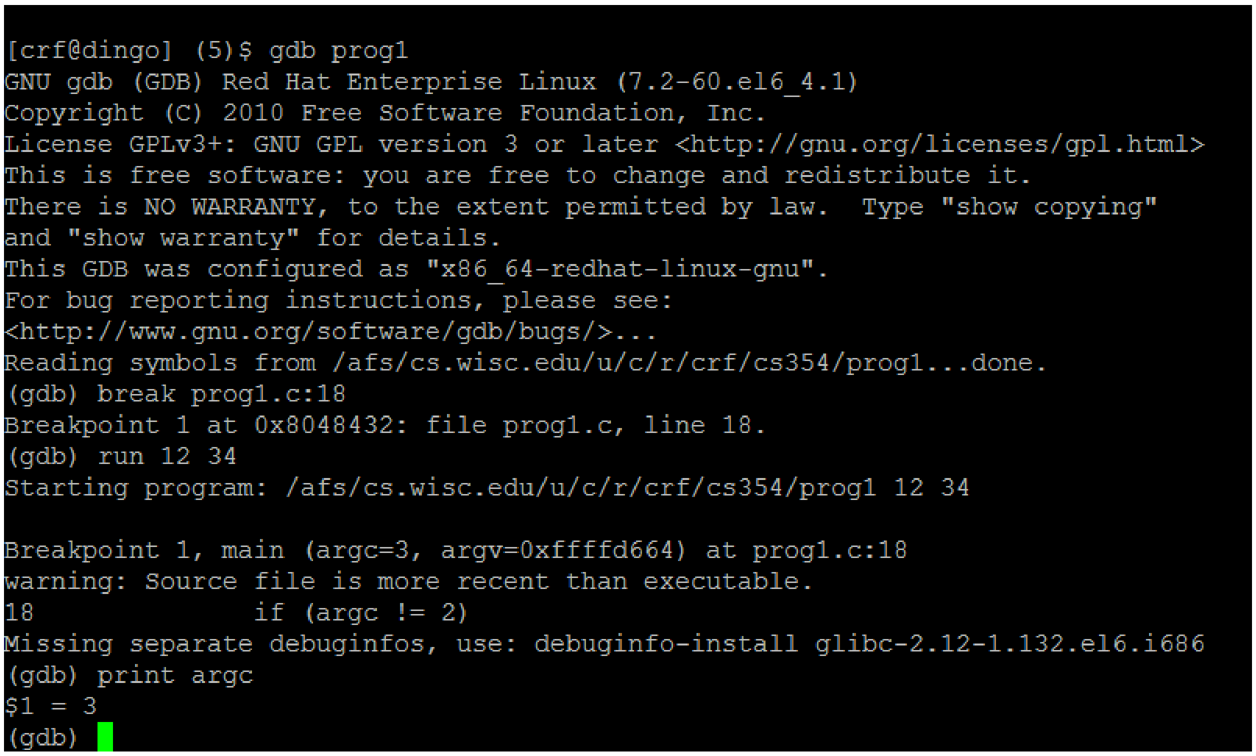


```
usage: prog1 number1 number2

[crf@dingo] (5)$ gdb prog1
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1...done.
(gdb) break prog1.c:18
Breakpoint 1 at 0x8048432: file prog1.c, line 18.
(gdb) run 12 34
Starting program: /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1 12 34

Breakpoint 1, main (argc=3, argv=0xffffd664) at prog1.c:18
warning: Source file is more recent than executable.
18              if (argc != 2)
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.132.el6.i686
(gdb) 
```

Note the line that says 18, and shows the conditional portion of the if statement. Also note two lines up that gdb stopped at breakpoint 1, which is the breakpoint we set at line 18. **Please ignore the warning about the source file: you won't see it when you run gdb. You also can safely ignore "Missing separate debuginfos …"**

Now that we stopped the program at line 18, we can do a bunch of things. One thing that we really want to be able to do is figure out what value is currently in argc. For any variable in our code, we can use the print command to determine what is currently stored inside it.

print argc



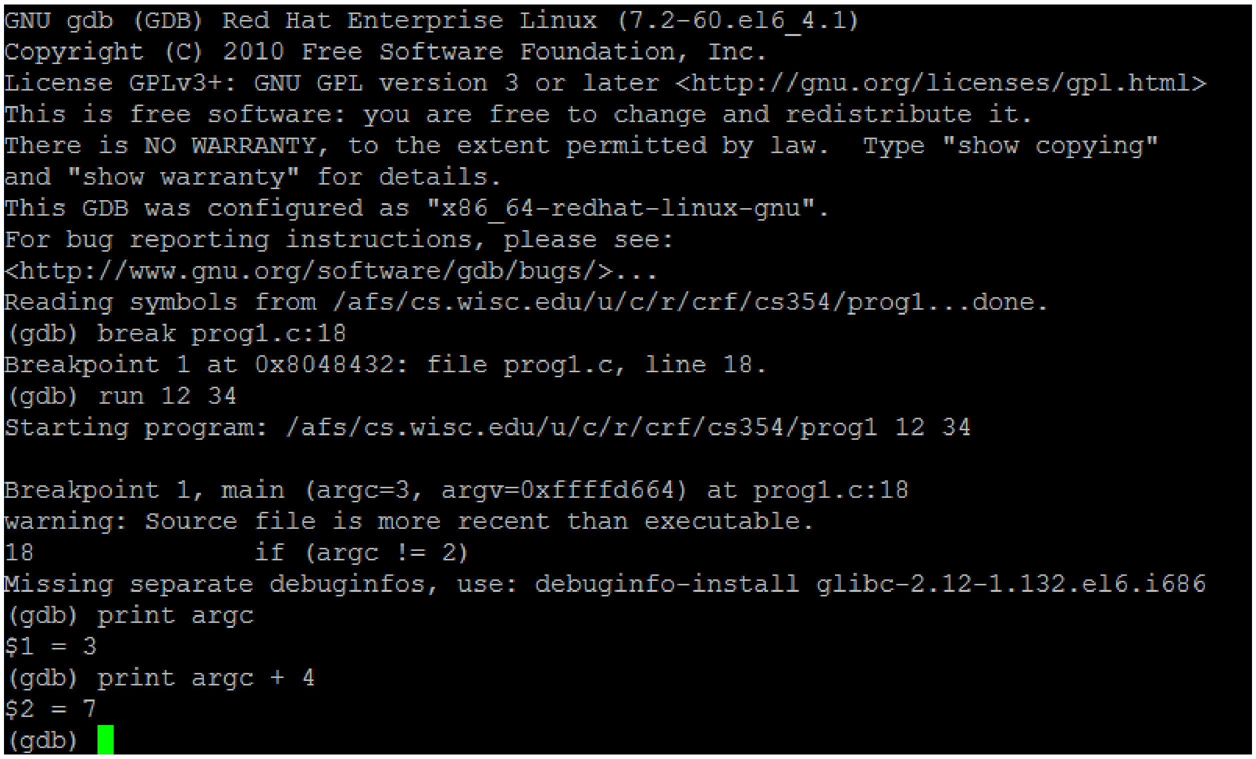
```
[crf@dingo] (5)$ gdb prog1
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1...done.
(gdb) break prog1.c:18
Breakpoint 1 at 0x8048432: file prog1.c, line 18.
(gdb) run 12 34
Starting program: /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1 12 34

Breakpoint 1, main (argc=3, argv=0xffffd664) at prog1.c:18
warning: Source file is more recent than executable.
18              if (argc != 2)
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.132.el6.i686
(gdb) print argc
$1 = 3
(gdb) 
```

Right under our print command, it says $1 = 3, which is gdb's way of telling us that argc = 3 (like we expected!). We can now go back to prog1.c, fix that bug, and recompile the code to search for new warnings.

While we're here however, there are also some cool and useful things we can do with the print command.

The print command can print out variables, but it's not just limited to variables. The print command can also print out the result of *expressions*. For example, if I wanted to know what argc + 4 was, I could ask gdb to print that and it would tell me. If I wanted to know something like the address of argc, I could ask gdb to print that for me too, in the exact same way you'd ask a C program for the address of something (&argc).

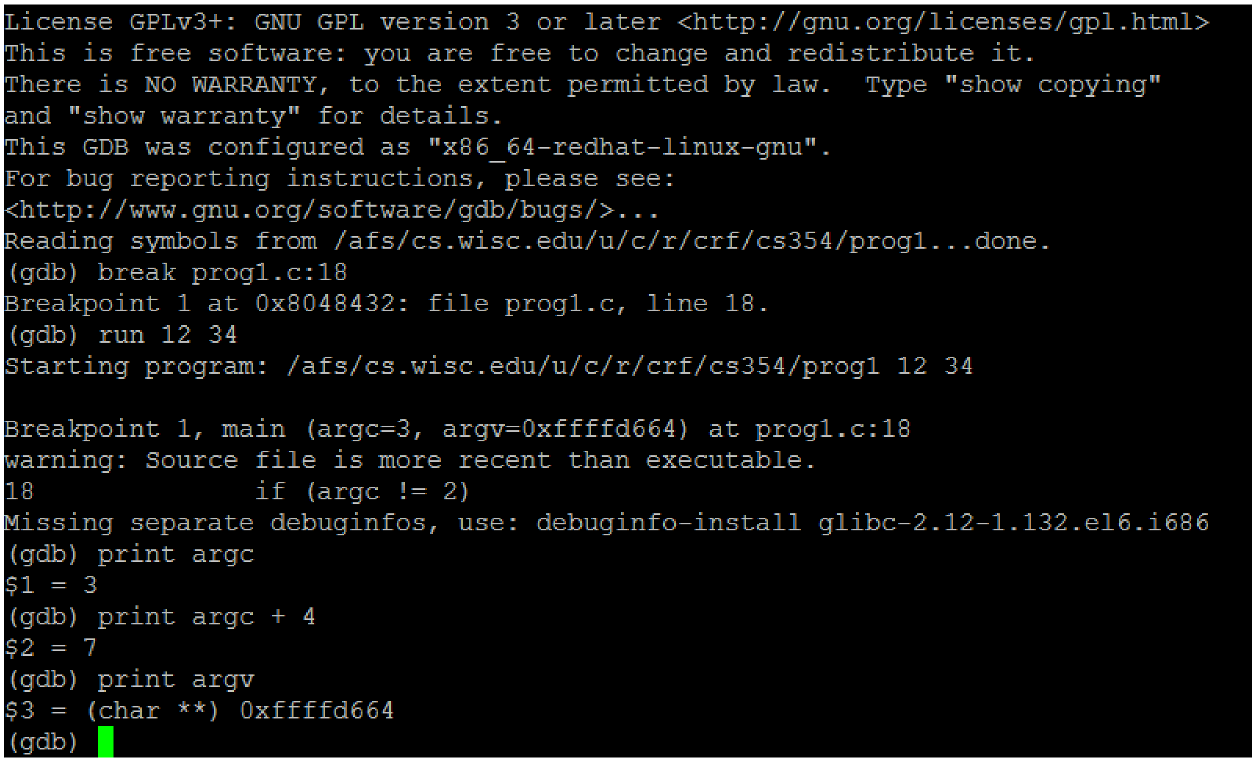
print argc + 4



```
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1...done.
(gdb) break prog1.c:18
Breakpoint 1 at 0x8048432: file prog1.c, line 18.
(gdb) run 12 34
Starting program: /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1 12 34

Breakpoint 1, main (argc=3, argv=0xffffd664) at prog1.c:18
warning: Source file is more recent than executable.
18              if (argc != 2)
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.132.el6.i686
(gdb) print argc
$1 = 3
(gdb) print argc + 4
$2 = 7
(gdb) 
```

We can also see what is inside arrays with print, although not quite the way you might expect. Say for instance we want to examine argv[][]. We can do print argv, but that will only return us an address to a pointer, like so:

print argv



```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1...done.
(gdb) break prog1.c:18
Breakpoint 1 at 0x8048432: file prog1.c, line 18.
(gdb) run 12 34
Starting program: /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1 12 34

Breakpoint 1, main (argc=3, argv=0xffffd664) at prog1.c:18
warning: Source file is more recent than executable.
18              if (argc != 2)
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.132.el6.i686
(gdb) print argc
$1 = 3
(gdb) print argc + 4
$2 = 7
(gdb) print argv
$3 = (char **) 0xffffd664
(gdb) 
```

We can examine memory locations using the x command, which is a command that you can look up in the manual for more information.

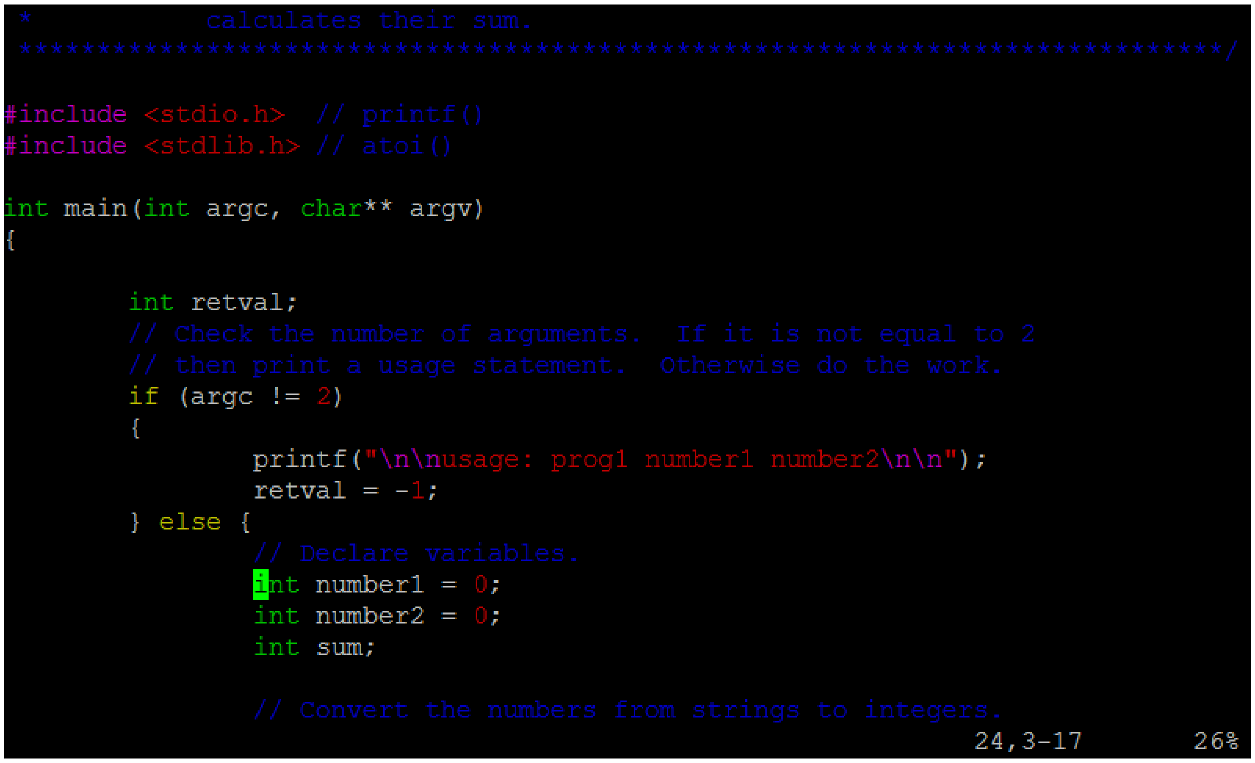
Single Stepping

Sometimes setting a breakpoint and examining values in variables is not enough to find the bug. Being able to run your program line by line is helpful so you can see how the variables change. This technique is especially useful for debugging problems with loops. It also is very helpful if you can't quite pinpoint where a bug is.

Let's use single stepping to find other errors. First quit out of the debugger by typing the following:

quit

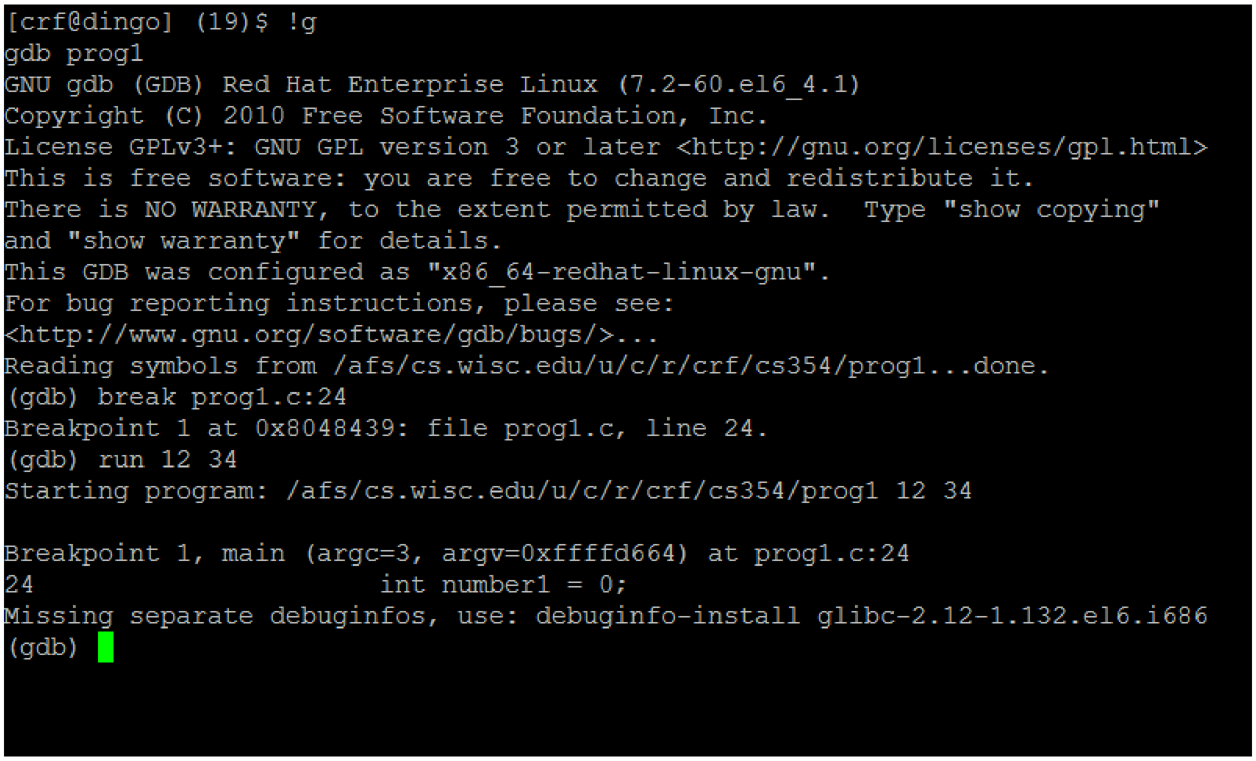
Go ahead and fix the argc bug that we just found and take a look at the code. Now that we know that argc is fixed (although, we should recompile and re-run the program to verify our fix works), let's put a breakpoint on the first statement in the else block:



```
 *           calculates their sum.
 ****************************************************************************/

#include <stdio.h>  // printf()
#include <stdlib.h> // atoi()

int main(int argc, char** argv)
{

        int retval;
        // Check the number of arguments.  If it is not equal to 2
        // then print a usage statement.  Otherwise do the work.
        if (argc != 2)
        {
                printf("\n\nusage: prog1 number1 number2\n\n");
                retval = -1;
        } else {
                // Declare variables.
                int number1 = 0;
                int number2 = 0;
                int sum;

                // Convert the numbers from strings to integers.
                                                              24,3-17       26%
```

If you haven't already, recompile your code and then start up gdb so that it is debugging prog1. Place a breakpoint on line 24 in prog1.c, and run the program with the same command line arguments as last time.

Here's what gdb outputted for me:



```
[crf@dingo] (19)$ !g
gdb prog1
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1...done.
(gdb) break prog1.c:24
Breakpoint 1 at 0x8048439: file prog1.c, line 24.
(gdb) run 12 34
Starting program: /afs/cs.wisc.edu/u/c/r/crf/cs354/prog1 12 34

Breakpoint 1, main (argc=3, argv=0xffffd664) at prog1.c:24
24                      int number1 = 0;
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.132.el6.i686
(gdb) 
```

We can make gdb execute the next line using the step command. By using the step command in conjunction with the print command and with breakpoints, we find several types of bugs.

You should step through the program line by line until you find where all the bugs are. Once they are found go back and fix and recompile, and continue the debugging process.

Debugging prog2.c

prog2.c contains a linked list program that has bugs. Using the techniques you learned above you should be able to find and squash the bugs in prog2.c

Debugging Assembly Language Code

gdb can also debug assembly language! The process is similar, except that breakpoints are set at addresses as opposed to the line.

The methodology doesn’t change much from debugging a C program.

Here is a list of useful commands:

- To set a breakpoint at an address addr: break *addr
- To step instruction by instruction: stepi
- To see the current state of all registers: info registers

How Do I Get Better at Debugging?

Learning to debug programs well is an art and is best taught by debugging programs. It takes a lot of practice to learn to do it well and you will get better as you write more programs and debug more code.