The GNU Project Debugger gdb

Debugging is the process of finding out where the bug in your code lies. It can fix crashes, remove any logical mistakes, and make your program more efficient. The gdb is a powerful debugging tool for C. Let's examine some of its features.

We'll cover the following

What is a symbolic debugger?

A debugger is a program that runs other programs, and provides the ability to control the execution of that program, for example stepping through one line at a time, and inspecting variables as the program is running. When your program crashes, and UNIX gives you a vague error message, a debugger will help you figure out where it is crashing, and (with the assistance of your brain) why it is crashing.

Why not use printf()?

A quick and easy method of debugging that doesn't require a separate debugger program, is to sprinkle your code with `printf()` statements that write to the screen, the values of various variables that you think are relevant and related to a crash (or other error). Some people call this adding "trace code" to your program.

The disadvantages of debugging using trace code are that you may need **many** `printf()` statements all over your program, and it becomes a nuisance to put them in, take them out, etc. Moreover a symbolic debugger can do a lot more stuff than simple trace code. It can:

- halt a program

- allow you to inspect variable values

- jump to an arbitrary line of code
- evaluate expressions
- restart from where you left off

You just get more fine-grained control by using a debugger. Finally, you can use the gdb debugger on a program that has already crashed, without having to re-start the program (you will see the state of the program, and its variables, at the time of the crash, allowing you to inspect the values of the variables, the location in the program of the crash, etc).

Compiling your program for gdb

To allow gdb to run your program, you must compile your program with a special compiler flag, `-g`. Here is a simple example of a program we will use to illustrate the gdb debugger `go.c`:

```
#include <stdio.h>
#include <stdlib.h>

char *getWord(int maxsize)
{
  char *s;
  s = calloc(sizeof(char), maxsize);
  printf("enter a string (max %d chars): ", maxsize);
  scanf("%s", s);
  return s;
}

void repeatWord(char *s, int n)
{
  int i;
  for (i=0; i<n; i++) {
    printf("%s\n", s);
  }
}

int *getVec(int vecsize)
{
  int *vec = malloc(sizeof(int)*vecsize);
  int i;
  for (i=0; i<vecsize; i++) {
    printf("enter value of vec[%d]:", i);
    scanf("%d", &vec[i]);
  }
  return vec;
}

void printVec(int *vec, int size)
```

```
{
  int i;
  printf("vec = {");
  for (i=0; i<size-1; i++) {
    printf("%d,", vec[i]);
  }
  printf("%d}\n", vec[size-1]);
}

int main(int argc, char *argv[])
{
  char *myWord = getWord(256);
  repeatWord(myWord, 3);

  int *myVec = getVec(5);
  printVec(myVec, 5);

  free(myWord);
  free(myVec);

  return 0;
}
```

go.c

Here is an example of what it does:

```
plg@wildebeest:~/Desktop/CBootCamp/code/debug$ gcc -g -o go go.c
plg@wildebeest:~/Desktop/CBootCamp/code/debug$ ./go
enter a string (max 256 chars): TheDude
TheDude
TheDude
TheDude
enter value of vec[0]:3
enter value of vec[1]:1
enter value of vec[2]:4
enter value of vec[3]:1
enter value of vec[4]:5
vec = {3,1,4,1,5}
```

Note that we compiled this using the `-g` compiler flag. This prepares the program to be used by the `gdb` debugger.

Stack frames

We've already talked about the **stack** when we talked about how variables can be allocated in C. This concept comes into play again, together with a concept called a **frame**. For now, we will go over the highlights and the main concept.

The stack (which we learned about already) is actually made up of **stack frames**. Each frame represents a function call. So as you call functions in your program, the number of stack frames increases, as the stack grows in size. As functions

return (and exit), the stack shrinks as the stack frames are popped off of the stack. Wikipedia has a very thorough description of how this works here.

Whenever a function is called in your program, an area of memory in the stack is set aside for that function (its stack frame). This memory chunk holds information like the storage space for variables declared in that function, the line number of the function that called the present function, and input arguments for the function. We can inspect these things in the gdb.

In particular, when your program crashes, you can ask gdb to tell you where it crashed, and you can ask for a **stack trace**, which is a list of what stack frames are on the stack.

Inspecting the stack using gdb

As an example, let's change the code above to introduce a bug that causes the program to crash. Let's comment out line 7:

```
//  s = calloc(sizeof(char), maxsize);
```

Now when we run our program this happens:

```
plg@wildebeest:~/Desktop/CBootCamp/code/debug$ gcc -g -o go go.c
plg@wildebeest:~/Desktop/CBootCamp/code/debug$ ./go
enter a string (max 256 chars): TheDudeAbides
Segmentation fault (core dumped)
```

So let's run the code from `gdb` now:

```
plg@wildebeest:~/Desktop/CBootCamp/code/debug$ gdb go
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 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-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/plg/Desktop/CBootCamp/code/debug/go...done.
(gdb) run
Starting program: /home/plg/Desktop/CBootCamp/code/debug/go
enter a string (max 256 chars): TheDudeAbides

Program received signal SIGSEGV, Segmentation fault.
_IO_vfscanf_internal (s=<optimized out>, format=<optimized out>,
    argptr=0x7fffffffe048, errp=0x0) at vfscanf.c:1095
1095    vfscanf.c: No such file or directory.
(gdb)
```

We launch with `gdb go`, and then we type the `gdb` command `run` to run the program from within `gdb`. As before, it crashes, with a slightly obscure error message. We can type the command `backtrace` to get a list of the stack frames:

```
(gdb) backtrace
#0  _IO_vfscanf_internal (s=<optimized out>, format=<optimized out>,
    argptr=0x7fffffffe048, errp=0x0) at vfscanf.c:1095
#1  0x00007ffff7a79fdd in __isoc99_scanf (format=<optimized out>)
    at isoc99_scanf.c:37
#2  0x000000000040067f in getWord (maxsize=256) at go.c:9
#3  0x00000000004007cc in main (argc=1, argv=0x7fffffffe258) at go.c:44
(gdb)
```

What we see is a numbered list of stack frames. The one labeled `#3` corresponds to the `main()` function, the one labeled `#2` the `getWord()` function, and the one labeled `#1` the `scanf()` function. The one labeled `#0` corresponds to some internal function used by `scanf()`.

This list of stack frames give you a picture of where you are in your program, and what functions have called what. Note that each stack frame entry provides both the name of the source file and the line number. This is very useful. We know for example that the crash occured at **line 9** of the **go.c** file (see the end of the entry labeled `#2`, that shows `at go.c:9`. We know from the entry labeled `#1` that the offending function call was to `scanf()`.

Now we can look at our code listing and try to figure out why `scanf()` failed. We've only given it two arguments, a format string, and a pointer to a buffer to hold the data. The format string looks fine ("%s") and so what we can do now is inspect the buffer using the `gdb` `print` command.

The feature of gdb that we will study next is the breakpoint system.