Introduction

GDB is a standard debugging program that is included by default in all Linux platforms. We are teaching it to you now so that when you start programming the rest of the projects (they are in C), when your code breaks you have a tool that you can use to diagnose the problem. In this exercise, we are providing to you two very simple C programs and asking you to debug the errors. (Being proficient at gdb is also a skill that can be put on a resume.)

If you show up to office hours needing help on your projects later this semester, *we expect you to have tried to debug your project yourself first using gdb.*

Instructions

gdb_example1

First, compile gdb_example1 with debugging options using gcc. Ignore any warnings for now.

```
gcc -o gdb_example1 gdb_example1.c -g
```

Try running the provided gdb_example1 using

```
>>> ./gdb_example1
```

You should get an segmentation fault. This is a very common problem. To easily figure out where this problem is, run

```
>>> gdb ./gdb_example1
```

GDB will show you the welcome prompt. To run the binary under gdb, type 'r' or 'run'. Gdb will automatically break at the segfault.

At this point, we want to know what caused the segfault. We can do one of several ways. We can either type 'list' at the gdb prompt (shows ± 10 lines around current stopped point), or open up the file in a text editor. Proceed either way to figure out what is going on around this segfault. Try to debug the error. After you have diagnosed the problem, make the required changes so that gdb_example1.c does not segfault. (Hint: don't make any changes that affect the functionality - if some piece of code is going cause a segfault, don't do it or delete the line.) If you can't figure out, an example fix is shown at the bottom of this README.

After we have made the fix, we need to compile and build this program again. Gcc provides a debugging option so that in gdb, you can access variables and functions by names from the source file (instead of memory addresses). To compile these c files with debugging symbols for gdb, gcc need to be run with '-g' option.

```
gcc -o <out_binary_name> <in_file_name>.c -g
```

In our case, we want

```
gcc -o gdb_example1 gdb_example1.c -g
```

Run gdb_example1 again. If your fix worked, there will be no segmentation fault. Congratulations, you have made your first GDB fix!

gdb_example2

Next, try your hand at gdb_example2. We will take a different approach to debugging this example than gdb_example1 for the sake of exposing you to additional gdb functionality. Again, first compile it (ignore

any warnings), then run it:

```
>>> gcc -o gdb_example2 gdb_example2.c -g
>>> ./gdb_example2
```

To debug,

```
>>> gdb gdb_example2
(gdb) run
```

We are interested in what are the values of the two variables causing the segfault. We see that gdb has broken on line 5, *ip = i; To see the values of the variables,

```
(gdb) print i
(gdb) print ip
```

From the value of ip, we can see that it is just pointing at some address, which may be causing the segfault. (Where was this address ever defined? Was this address ever defined?) We also see that this segfault is happening during a function call. To have gdb list the full function stack,

```
(gdb) info stack
```

We can see now that setInt() is called by main(), so it seems that main also called this function with invalid arguments. To see the arguments to the current function call,

```
(gdb) info args
```

We have a lot of information about the state of the binary when the segfault occurs now. To list them, we know:

- The value of the variables causing the segfault
- The place in code where the function was called
- The value of the function arguments of the current function

Make any fixes to allow the binary to run. Compile the binary (remember debugging options!) and run it to verify that your fixes solved the segfault.

Debug a Program

BuggyBST is provided as a final exercise to play around with GDB. An excellent way to learn how to use gdb is to google 'gdb cheatsheet' and read over the different commands that gdb provides. Some very useful ones are: 'gdb –args' next/step/continue breakpoints conditional breakpoints watchpoints examining memory 'display'

Compile buggyBST.c and ignore all warnings for the sake of this exercise - use gdb to quickly find all of the errors. Make any changes to the line required to fix the segfault. *Hint: Most of the errors can be fixed by one character changes - they should seem like you are fixing a typo.*

Deliverables

Make the fixes to gdb_example1, gdb_example2, and buggyBST. After you have saved your changes, submit the following in a .tar.gz file.

- gdb_example1.c
- gdb_example2.c
- buggyBST.c

```
//******************** Example Fixes ***************************//

//gdb_example1.c fix:
void print_scrambled(char *message)
{
  int i = 3;
  if (message != NULL) {            // check for NULL pointer
      do {
        printf("%c", (*message)+i);
      } while (*++message);
      printf("\n");
  }
}

//*****************************************************************//

//gdb_example2.c fix:
void setint(int* ip, int i) {
    *ip = i;
}

void write_message(char *message) {
    char buffer[100];

    memset(buffer, '\0', 100);
    strcpy(buffer, message);
    printf("%s\n", buffer);
}

int main() {
    int a;
    char message[] = "Look, this seems like an innocent message!";

    setint(&a, 10);
    printf("%d\n", a);

    write_message(message);

    int *b;
    int c;
    b = &c;             // point B to a valid integer object
                        // instead of leaving it to point to whatever
                           is
                        // leftover on the stack

    setint(b, 20);
    printf("%ld\n", *b);
    printf("This may or may not have crashed. You are lucky if you see
       this.");
    return 0;
}

//*****************************************************************//
```