Lab 5: A Brief Tutorial on GDB

For this lab, create a new directory named `lab5` under your `cs449` directory and create your program there:

```
mkdir lab5
cd lab5
```

For this lab, you can get the starter file (`badprog.c`) to your directory using the following command:

```
cp /afs/cs.pitt.edu/usr0/tkosiyat/public/cs0449/badprog.c .
```

For those who works on Ubuntu on your own computer, the starter file is also available on the CourseWeb.

1 GNU Debugger

As your programs getting larger, debugging for errors would become much more challenging. The nightmare of segmentation faults might be caused by bugs so minor that finding them would be almost impossible. This is when `gdb` comes to your aid. `gdb` is the GNU Debugger that allows you analyse the runtime of your program. It provides facilities such as inspecting variables, function calls and examining the content of a stack frame. Well suited for C and C++, GDB is especially effective in debugging segmentation faults and other runtime errors.

We would begin with `badprog.c` which is a program that finds the maximum value in an array. First, compile and run the program as usual using the following command:

```
gcc -o badprog badprog.c
./badprog
```

The array under test is `[17,21,44,2,60]`. After you compile and run the program, the output should be:

```
max value in the array is 17
```

which is incorrect. You might have spotted where the error is but let's try using `gdb` to figure it out. First, let's compile the program again using the following command:

```
gcc -g -fvar-tracking -o badprog badprog.c
```

Running `gcc` with the `-g` flag puts additional information in the resulting executable that would be used by `gdb` to debug. For the `-fvar-tracking`, this is optional. Some version of `gdb` have a bug where the value of expression in C is changed by `gdb` still reports the old value. So, to gbe safe, use the flag `-fvar-tracking` as well. Now, let's debug this program using the following command:

```
gdb badprog
```

You should see a copyright disclaimer. The `gdb` prompt `(gdb)` is where you would type gdb-specific commands.

Commands in gdb

Here are some commonly used commands in gdb. You can learn more about all of the commands by typing help or on a specific command by typing help command_name.

Command	Shortcut	Description
help		Get help on a command or topic
set args		Set command-line arguments
run	r	Run (or restart) a program
quit	q	Exit gdb
break	b	Place a breakpoint at a given location
continue	c	Continue running the program after hitting a breakpoint
backtrace	bt	Show the function call stack
next	n	Go to the next line of source code without entering a function call
step	s	Go to the next line of source code, possibly entering a new function
nexti	ni	Go to the next instruction (assembly) without entering a function call
stepi	si	Go to the next instruction (assembly) possibly entering a new function
print	p	Display the value of an expression written in C notation
x		Examine the content of a memory location (pointer)
list		List the source code of the program
dissemble	disas	List the machine code of the program

Example

Let's try the command b main and press enter as shown below:

```
(gdb) b main
Breakpoint 1 at 0x4005b1: file badprog.c, line 26.
```

This command creates a breakpoint at the main function of the badprog.c. A breakpoint is a point in your code where execution pauses. It is a useful way of examining specific portions of the program. You can put breakpoints on a line, on a function, or at an address. For mroe information on breakpoints go to http://www.delorie.com/gnu/docs/gdb/gdb_29.html.

Enter the command r which means run. This would run the program until it hits the breakpoint and stops.

```
(gdb) r
...
Breakpoint 1, main () at badprog.c:26
26        {
```

The program stops on line 27 and shows the code at that point. From the program, we can observe that much of the work is done in that findAndReturnMax function. Let's put a breakpoint there:

```
(gdb) b findAndReturnMax
Breakpoint 2 at 0x400544: file badprog.c, line 4.
```

Hit the continue command `c` to get into the `findAndReturnMax` function. This would continue the execution of the program until it hits the breakpoint.

```
(gdb) c
Continuing.

Breakpoint 2, findAndReturnMax (array1=0x7fffffffdeb0, len=5, max=17)
    at badprog.c:4
4        {
```

You may notice that `gdb` also display the values of arguments of function `findAndReturnMax`. In the function, you could check the contents of variables using the `p` command:

```
(gdb) p array1
$1 = (int *) 0x7fffffffdeb0
(gdb) p *array1
$2 = 17
(gdb) p len
$3 = 5
(gdb) p max
$4 = 17
```

Note that if a value is a pointer, `gdb` will also show what type of pointer together with its value. By using the `print` (`p`) command, it allows use to examine the content of variables as they are updated. To move to the next line, use the `n` command. The following is an example of using a series of `n` command and then check the value of the variable `max`:

```
(gdb) n
7                if(!array1 || (len <= 0))
(gdb) n
12                max = array1[0];
(gdb) n
14                for(i = 1; i <= len; i++)
(gdb) n
16                        if(max < array1[i])
(gdb) n
18                                max = array1[i];
(gdb) n
14                for(i = 1; i <= len; i++)
(gdb) p max
$5 = 21
```

As the variables get updated, check their contents to see if it is what to expect. When done with the function, use the `finish` command to exit the function:

```
(gdb) finish
Run till exit from #0  findAndReturnMax (array1=0x7fffffffdeb0, len=5, max=21)
    at badprog.c:14
0x00000000004005f6 in main () at badprog.c:31
31                if(findAndReturnMax(arr, 5, max1) != 0)
Value returned is $6 = 0
```

This executes the function and returns to the caller (`main` function). You can see that the function return 0 which shows that the function completed successfully.

Logically, the `max` variable in the `main` function should hold the max value as it was passed as an argument in the `findAndReturnMax` function. Let's examine the value of `max` in the `main` function:

```
(gdb) p max
$7 = 17
```

The `max` variable still has the wrong value. The function `findAndReturnMax` supposes to update the variable `max` to the maximum value in the array. There must be something wrong. **Note** that C is call by value. So, if we want the function `findAndReturnMax` to modify the value of `max`, we have to send the address of `max` variable. To do so, we need to modify the signature of the function `findAndReturnMax` to take the address (pointer) of integer as shown below:

```
int findAndReturnMax(int *array1, int len, int *max)
```

Note that you also need to modify the the expression `max` inside the function to `*max` (dereference). So, let's exit `gdb` using `Ctrl-D` and modify the code. **Note** that you also have to change how the function `findAndReturnMax` is called in the `main` function to send the address of `max` as shown below:

```
if(findAndReturnMax(arr, 5, &max) != 0)
```

That is, your program should look like the following:

```
#include <stdio.h>

int findAndReturnMax(int *array1, int len, int *max)
{
        int i;

        if(!array1 || (len <= 0))
        {
                return -1;
        }

        *max = array1[0];
```

```
        for(i = 1; i <= len; i++)
        {
                if(*max < array1[i])
                {
                        *max = array1[i];
                }
        }

        return 0;
}

int main(void)
{
        int arr[5] = {17, 21, 44, 2, 60};

        int max = arr[0];

        if(findAndReturnMax(arr, 5, &max) != 0)
        {
                printf("strange error\n");
                return 1;
        }

        printf("max value in the array is %i\n", max);

        return 0;
}
```

After you finish modifying the code, compile it and run to see whether it gives you the right maximum value. In my case, the result is shown below:

```
max value in the array is 32767
```

Can you spot why? If you cannot, why don't you use `gdb` and try to examine the function `findAndReturnMax` closely to see what is the problem. When you find the problem, fix it, compile it, and run it to verify that it works correctly.

2 Infinite Loop

For this section, you can get the starter file (`infiniteloop.c`) to your directory using the following command:

```
cp /afs/cs.pitt.edu/usr0/tkosiyat/public/cs0449/infiniteloop.c .
```

Compile and run it and you will see that the program does not terminate. So, let's compile it for `gdb` and run it under `gdb`:

```
gcc -g -fvar-tracking -o infiniteloop infiniteloop.c
gdb infiniteloop
```

```
...
(gdb) r
...
```

Again, under `gdb` it does not return back to the `gdb` prompt (`(gdb)`). So, press `Ctrl-C` to stop the program.

```
^C
Program received signal SIGINT, Interrupt.
foo (dest=0x7fffffffde90 "aWV", src=0x7fffffffdeb0 "abcdefg")
    at infiniteloop.c:8
8               while(src[i] != '\0')
```

Note that some of you may get the following output after press `Ctrl-C`:

```
^C
Program received signal SIGINT, Interrupt.
0x000000000040058f in foo (dest=0x7fffffffde90 "aWV",
    src=0x7fffffffdeb0 "abcdefg") at infiniteloop.c:10
10                      dest[i] = src[i];
```

Now type in the command `where` which will tell you where it was before you press `Ctrl-C`:

```
(gdb) where
#0  0x000000000040058f in foo (dest=0x7fffffffde90 "aWV",
    src=0x7fffffffdeb0 "abcdefg") at infiniteloop.c:10
#1  0x00000000004005db in main () at infiniteloop.c:19
```

The above information tells you that, it was at line 10 in function `foo` (some of you may get line 8). This gives you a hint that it may stuck in a look somewhere around line 10 (or 8). Use the `list` command to see the code:

```
(gdb) list
3
4       void foo(char *dest, char *src)
5       {
6               int i = 0;
7
8               while(src[i] != '\0')
9               {
10                      dest[i] = src[i];
11               }
12       }
```

So, fix the `infiniteloop.c` so that it does not run indefinitely. Note that the purpose of function `foo()` is to copy string from `src` to `dest`.

What to Hand In

First, let us go back up to our cs449 directory:

```
cd ..
```

Now, let us first make the archive. Type your username for the USERNAME part of the filename:

```
tar cvf USERNAME_lab5.tar lab5
```

And then we can compress it:

```
gzip USERNAME_lab5.tar
```

Which will produce a `USERNAME_lab5.tar.gz` file.

If you work on `cs449.cs.pitt.edu` (`thoth`) you can skip to the next section. **If you use Ubuntu your own machine, you need to transfer the file to `cs449.cs.pitt.edu` first**. This can simply be done by a command line. For example, assume that your username is `abc123` and you are in the same directory as the file `abc123_lab5.tar.gz`. To transfer the file to `cs449.cs.pitt.edu` use the following command:

```
scp abc123_lab5.tar.gz abc123@cs449.cs.pitt.edu:.
```

The above command will copy the file to your home directory in `cs449.cs.pitt.edu`. If you want to copy it to your `private` directory, use the following command:

```
scp abc123_lab5.tar.gz abc123@cs449.cs.pitt.edu:./private/.
```

Copy File to Submission Directory

We will then submit that file to the submission directory:

```
cp USERNAME_lab5.tar.gz /afs/cs.pitt.edu/public/incoming/CS0449/tkosiyat/sec1
```

Once a file is copied into that directory, you cannot change it, rename it, or delete it. If you make a mistake, resubmit a new file with slightly different name, being sure to include your username. For example `USERNAME_lab5_2.tar.gz`. **Check the due date of this lab in our CourseWeb under Labs/Recitations**.