CpSc 102
Lab 1: Debugging Tools

Goals

This lab will introduce you to **gdb**, a tool that can be used to debug programs that have run-time errors. You may find this to be a useful tool as you begin writing more complicated programs than the ones you have seen thus far, and encounter run-time errors such as "segmentation faults" or "bus errors". After you complete this lab, you should continue to experiment with gdb so that you can become more familiar with it - especially if you have programs that have run-time errors. It is very likely that at some point during this semester, you will need to use gdb to try to figure out the point at which your program fails.

The other tool that will be introduced is called **valgrind**. This tool can be used to detect a number of common problems encountered when using pointers. One of those problems is memory leaks. A *memory leak* is said to have occurred when:

1. the last pointer to a malloc'd object is reset, or
2. the last pointer to a malloc'd object is a local variable in a function from which a return is made.

In these cases, the malloc'd memory is no longer accessible. Excessive memory leaking can lead to poor performance and, in the extreme, program failure. Therefore, C programmers must recognize when the last pointer to malloc'd storage is about to be lost and use the `free()` function call to release the storage before it becomes impossible to do so.

One of the files you will copy over to your account is a text file called answers.txt. There are 7 questions for you to answer as you work through the examples on the following pages. That is the only file you will submit when you are finished answering the questions. Use the handin page at http://handin.cs.clemson.edu to submit your work.

Assignment Part I

gdb Debugger

The gdb debugger is an effective interactive tool that will allow you to run a program, stopping at predefined break points that you set; print the value of variables, lines of code, etc.; continue executing to the next break point, or line by line; and find the statement at which a program suffered a fatal error.

In order to use gdb (or any other debugger) on your programs, you must instruct the compiler to include *debugging symbols* in the executable. Otherwise, the compiler leaves out these symbols to reduce the size of the executable files. With the gcc compiler, the -g switch turns on debugging symbols. In other words, if you have a program called `prog1.c`, you would type the following when compiling:

```
gcc -g -Wall prog1.c
```

Some of the commonly used gdb commands are shown in the table below:

gdb ./a.out	load the program "a.out" in the current working directory and start the debugger
gdb –tui ./a.out	load the program "a.out" in the current working directory and start the debugger; using the **-tui** splits the screen showing the code in the upper half and the gdb prompt in the lower half
break main (or: **b main**)	cause execution to pause at the start of the function "main"
break 32 (or: **b 32**)	cause execution to stop at line 32 (or whatever line number you specify)
run (or: **r**)	start execution of the currently loaded program
r > outputFile.txt	start execution of the currently loaded program, redirecting the output to the file specified (could be useful for debugging programs that produce an image file, e.g a .ppm image)
n (or **next**)	execute the next line of source code
c (or **continue**)	continue without stopping to the next breakpoint, program termination, or error
p x (or **print x**)	print (to the screen) the current value of the variable x (or whatever variable name is specified)
d x (or **display x**)	display the current value of x at each gdb command prompt
q (or **quit**)	quit gdb
r	restart the currently running program using the previous command line

A more in-depth list of commands may be found here: http://people.cs.clemson.edu/~chochri/102/Notes/gdb-basics.pdf or by searching online.

First gdb example:

Copy the program below (and the other files needed for this lab) into your account by executing the following command:

```
cp /group/course/cpsc102/public_html/F15Labs/F15/lab01/Public/* .
```

```
/* factorial.c
   example to use with gdb

   no segfault, but the program does not work
   - - why??
*/

#include <stdio.h>

long factorial(int n);

int main(void) {
   int n;
   long val;

   printf("enter a number: ");
   scanf("%i", &n);
   val = factorial(n);

   printf("val is %li\n\n", val);
}

long factorial(int n) {
   long result = 1;

   while (n--) {
      result *= n;
   }

   return result;
}
```

1. With the editor of your choice, take a look at the `factorial.c` program.
2. Compile it: `gcc -g factorial.c` (The `-g` option adds debugging symbols to the executable, allowing for the use debugging tools)
3. Try to run the program the regular way without gdb: `./a.out`
4. Use 3 for the input number – you should get a 6 as the result (Remember that factorial of 3, written as 3! means 3 * 2 * 1 which should be 6). What result do you get?
5. Try it again with 5 for the input number – you should get 120 as the result (5! = 5 * 4 * 3 * 2 *1 = 120). What result do you get?
6. Start it with gdb: `gdb -tui ./a.out`
7. Using the `-tui` option above, you will see the gdb screen broken into two areas: the top part of the window shows the code for a portion of the program; the bottom part shows the gdb prompt where you will type your gdb commands. Type the following to set a break point at the point where the factorial function is entered: `break factorial`
8. `run` (this starts running the program up to the point in the program where it is waiting for the user to enter a number)
9. Enter a 3 and hit the return key. You can see, in the top part of the window, that the program executed the line of code where the input number was taken in and ran up to the breakpoint at the factorial function.
10. `print n` You will see that n does have the value of 3, the value sent to the factorial function.
11. `print result` You will see that something like `$1 = 140737354130120` prints to the screen instead of `1`. The debugger stops at the line preceding the current line of code shown, so when you try to print `result`, at that point, the value of `1` hasn't been assigned yet. In other words, the line of code that is highlighted is the line that *will be executed next*.
12. `next`
13. `print result` Now the value for `result` should be showing as `1`.
14. `next` (You can type an `n` instead of the whole word `next` – either will work. Same with `p` instead of `print`.)
15. Step through the loop using `next` and `p result` and `p n` to see what's happening for each line. Stop when you reach the end of the loop.
16. When line 30 is highlighted in the top part of the window

    30 return result;

    type `p result`. Shouldn't `result` be 6?? Were you able to spot the problem when you were stepping through the loop? What's going on??

Second gdb example:

```
/* segFault.c
   program with a segfault
   to be used with gdb
*/

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

int main(void) {
   char *buf;

   buf = malloc(1<<31);

   fgets(buf, 1024, stdin);
   printf("\n%s\n\n", buf);

   return 0;
}
```

1. Compile this program: `gcc -g segFault.c`
2. Try to run it normally – it will wait for you to type words as input and it is supposed to print them back out to you.
3. You will see a segfault occur.
4. Start it up with gdb: `gdb –tui ./a.out`
5. Type: `run` and you should see the following:

```
Program received signal SIGSEGV, Segmentation fault.
_IO_getline_info (fp=0x7ffff7dd4340, buf=0x0, n=1023, delim=10, extract_delim=1, eof=0x0)
    at iogetline.c:91
(gdb)
```

   This shows that a SIGSEV signal was received from the operating system, showing an attempted access to an invalid memory address.

6. Type: `backtrace` and you should see the following:

```
#0  _IO_getline_info (fp=0x7ffff7dd4340, buf=0x0, n=1023, delim=10, extract_delim=1,
       eof=0x0) at iogetline.c:91
#1  0x00007ffff7a8906b in _IO_fgets (buf=0x0, n=<optimized out>, fp=0x7ffff7dd4340)
       at iofgets.c:58
#2  0x00000000004005f7 in main () at segFault.c:14
```

   A bracktrace is a summary of how your program got to where it is, showing a list of all the function calls that lead to the crash. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero) followed by its caller (frame one) and on up the stack. You can see that there was a getline function in the current frame (frame #0) that was called from a fgets function (frame #1), which was called from the main function (frame #2).

   Since we are only interested in our code, which is in the main function, we want to switch to stack frame #2 and see where the program crashed.

7. Type: `frame 2`
   After doing that you will see this in the bottom part of the gdb window

```
#2  0x00000000004005f7 in main () at segFault.c:14
```

   and this line highlighted in the top part of the gdb window, indicating that is where the segfault occurred.

```
14              fgets(buf, 1024, stdin);
```

8. Try printing the value of `buf` and see what you get: `p buf`
   The value of `buf` is `0x0`, which is a NULL pointer. This is not what we want – `buf` should be pointing to the memory that was allocated in line 12. So, we'll kill the currently running invocation of the program and then set a break point at line 12, and then run again:
9. `kill` and answer `y` for yes
10. `break segFault.c:12`
11. `run`
12. `p buf` – the value of `buf` is `0x0`, which makes sense that it's not some address value because malloc hasn't executed yet
13. `n`
14. `p buf` – the value of `buf` is still `0x0`, which doesn't make sense this time because malloc has executed. Why would this be???

Assignment Part II

Valgrind debugging utility

Two cautions about Valgrind:

1. it isn't perfect -- it won't catch everything,
2. it is SLOWWWWW. Your program may run 10-20 times slower under Valgrind

But you don't need to run your program every time under Valgrind. You can use it periodically during the debugging stage for the program but discontinue its use when things have stabilized.

```
/* memoryLeak.c
   example to use for valgrind

   this program will reserve a block of memory
   large enough to hold 100 characters
*/

#include <stdlib.h>

int main(void) {
   char *x = malloc(sizeof(char) * 100);
   return 0;
}
```

1. Compile the program above `gcc -g memoryLeak.c`
2. Type the following: `valgrind --tool=memcheck --leak-check=yes ./a.out`
3. You should see the following:

```
==16204==  Memcheck, a memory error detector
==16204==  Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==16204==  Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==16204==  Command: a.out
==16204==
==16204==
==16204==  HEAP SUMMARY:
==16204==  in use at exit: 100 bytes in 1 blocks
==16204==  total heap usage: 1 allocs, 0 frees, 100 bytes allocated
==16204==
==16204==  100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==16204==  at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==16204==  by 0x400505: main (memoryLeak.c:11)
==16204==
==16204==  LEAK SUMMARY:
==16204==  definitely lost: 100 bytes in 1 blocks
==16204==  indirectly lost: 0 bytes in 0 blocks
==16204==  possibly lost: 0 bytes in 0 blocks
==16204==  still reachable: 0 bytes in 0 blocks
==16204==  suppressed: 0 bytes in 0 blocks
==16204==
==16204==  For counts of detected and suppressed errors, rerun with: -v
==16204==  ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

The three lines in the center shows how much memory is lost and where in the program the lost memory was allocated:

```
==16204==  100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==16204==  at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==16204==  by 0x400505: main (memoryLeak.c:11)
```

There will be times when the `--leak-check=yes` option will not result in showing you all memory leaks. To find absolutely every unpaired call to free or new, you'll need to use the `--show-reachable=yes` option. Its output is almost exactly the same, but it will show more unfreed memory. More of this tutorial for using Valgrind can be found here:
http://www.cprogramming.com/debugging/valgrind.html