**CPSC 1071 Lab 6   
gdb and Valgrind**

**Lab Objectives**

* Learn to use gdb to debug run-time errors
* Use Valgrind to help debug a program and understand memory management

**Introduction**

In CPSC 1070/1071, we treat errors and warnings as having the same severity, since warnings often indicate that the program will behave in a buggy or unsafe way. (There are exceptions, with warnings that are harmless, but we will not encounter these cases here.) The -Wall turns on additional warnings that can help catch bugs.

This lab gives an introduction to gdb and also introduces you to the Valgrind utility.

**Getting the files for this lab:**

Create a lab6 directory, move to the lab6 directory, and copy the files for this lab using:

lab1071copy  6

Your lab6 directory should now contain the files:

***answers.txt***   
***broken1.c***   
***broken2.c***   
***data.txt***   
***leak.h***   
***leak.c***

**Task 1: Using gdb to Identify Execution-Time Errors**

The *gdb* debugger is a crude but effective interactive debugger that will allow you to set breakpoints where execution will be halted within a program; print the value of a simple variable, array or structure; single step through a program; and find the statement at which a program suffered a fatal error (crash).

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 sizes of the executable files. With the gcc compiler, the *-g* switch turns on debugging symbols. We will generally use this switch in CPSC 1070/1071.

The major gdb commands are:

|  |  |
| --- | --- |
| **gdb ./a.out** | load the program "a.out" in the current working directory and start the debugger |
| **gdb ./sample** | load the program "sample" in the current working directory and start the debugger |
| **b main** (*or* break main) | cause execution to pause at the start of the function "main" |
| **b 32** (*or* break 32) | cause execution to pause at line 32 |
| **b sub1.c:20** | cause execution to pause at line 20 in the source file "sub1.c" |
| **r** (*or* run) | start excution of the currenly loaded program |
| **r  20 30  < in.txt  > out.txt** | start excution of the currenly loaded program, passing in the argv[ ] values "20" and "30", and redirect input from the file "in.txt" and output to the file "out.txt" |
| **s** (*or* step) | execute the next line of source code |
| **n** (*or* next) | execute the next line of source code stepping OVER function calls |
| **c** (*or* continue) | continue without stopping to the next breakpoint, program termination or error. |
| **p x** (*or* print x) | print the current value of the variable x |
| **d x** (*or* display x) | display the current value of x at each gdb command prompt |
| **q** | quit gdb |
| **r** | restart the currently running program using the previous command line |
| **w** (*or* where) | display where the program is executing or aborted and show the trace of currently active procedure calls |

An effective way to use gdb is to open three windows. Edit the source code in one window, run gdb in the second, and compile in the third.

**Task 1 - identifying execution-time errors in broken1.c**

1. Note that the previous copy operation copied a program called *broken1.c* and a data file, *data.txt*.
2. Compile *broken1.c* creating the executable file *broken*.

***gcc -o broken -Wall -g broken1.c***

1. Now try running the program using the file *data.txt* as input, e.g.:

***./broken < data1.txt***

Note the use of the file redirect for input.

You should get the message "Segmentation fault". This and "Bus Error" are the two most common run-time errors. Note that no indication is given as to where the program failed. You can imagine in a large program the message is pretty useless in trying to locate the problem.

1. Assuming you compiled broken1.c using the "-g" flag, initiate the program using gdb, i.e.

***gdb broken***

At the gdb prompt, enter

***r  <  data.txt***.

gdb should report where in the program the seg fault occured. It should look something like this:

Program received signal SIGSEGV, Segmentation fault.

0x00000000004005d7 in main (argc=1, argv=0x7fffffffeaf8)

at broken1.c:28

28 if ((table[ndx++] < 0) || (count < ndx))

Let's use the *print* or *p* command to print the current value of count. **Exercise:**In your *answers.txt* file, enter answers to the Task 1 Questions.

**Task 2 - identifying execution-time errors in broken2.c**

Next we will turn our attention to broken2.c.

* 1. As before, compile broken2.c to produce the executable file broken using the "-g" option.

***gcc -o broken -Wall -g broken2.c***

Try running the program using the command:

***./broken < data.txt***

The program will probably seg fault. If it does not seg fault, try running it again. NOTE: if broken still does NOT seg fault when you run it the second time, please notify the lab instructor and wait for the lab instructor's response before proceeding.

* 1. As in Task 1, bring up broken under gdb. Enter the gdb command:

***gdb broken***

* 1. **Breakpoints**

Before we begin running the executable, we would like to set a *breakpoint* so *gdb* will pause the program to allow us to inspect our variables. Let's set a breakpoint at line 32.

***b 32***

You should get a response similar to the following:

Breakpoint 1 at 0x40073a: file broken2.c, line 32.

**Note:** If you received the response, *No symbol table is loaded. Use the 'file' command* this means you did not include the *-g* option when you compiled the program.

* 1. **Walking Through the Code**

To start the debugging process, run the program:

***run 10***  
OR   
***r 10***

(Many *gdb* commands can be shortened by using their first letter).

The *run* command tells *gdb* to begin executing the program. It should stop at the first breakpoint, or when the program stops by itself, whichever comes first.

*gdb* should have printed out the line at the breakpoint. It should look something like this:

Breakpoint 1, main (argc=2, argv=0x7fffffffeaf8) at broken2.c:32

32 size = sizeof(data) / sizeof(int);

Let's continue by moving to the next line using *next* or *n*.

***n***

This brings us to line 33.

33 assert(n <= size);

Since line 33 is just an assert statement, use the ***n*** command to move to the next line.

***n***

We should now be at the following line:

copyData(data, copy, n);

This line looks interesting, but if we use the *next* command again, we will skip over the *copyData()* function. The *step* or *s* command can help us here. *step* allows us to enter into the body of the function, letting us go a level deeper. In this case, it would step into the *copyData()*function. If we were already at the deepest level, *step* would function the same as *next*.

Enter the command

***s***

We should now be at line 57 , inside the *copyData()* function. It should look something like this:

copyData (data=0x7fffffffe9c0, copy=0x0, n=10) at broken2.c:57

Notice in the header of the *copyData()* function, the value of the copy parameter is 0x0. We can also view the current state of our parameters and variables by using the ***print*** command.

***print copy***

This is what I get:

$1 = (int \*) 0x0

0x0 is an address generally reserved to represent NULL values, implying that *copy* is a NULL value.

The program has not seg-faulted yet because we have not attempted to use *copy*. When we execute this line of code, we should see the seg-fault. Enter the command ***n***.

Program received signal SIGSEGV, Segmentation fault.

0x0000000000400861 in copyData (data=0x7fffffffe9c0, copy=0x0,

n=10) at broken2.c:59

59 \*(copy + i) = \*(data + i);

Enter the gdb command "quit" to end this gdb session. <b<>Exercise:Open *broken2.c* and fix the the problem that is causing the seg-fault. Compile and execute the code to make sure that it works.</b<>

**Sharpening your gdb debugging skills**

Some helpful gdb tutorials:

<https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>   
<https://www.tutorialspoint.com/gnu_debugger/index.htm>   
<http://www.dirac.org/linux/gdb/>   
<http://cs.baylor.edu/~donahoo/tools/gdb/tutorial.html>   
<http://www.thegeekstuff.com/2010/03/debug-c-program-using-gdb/>

**Memory Leaks and Valgrind**

The Java language provides for "automagic" garbage collection in which storage for an object is magically reclaimed when all references to an object have gone out of existence.

C provides no such mechanism.

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 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.

**Task 3: Valgrind debugging utility**

Valgrind is a debugging tool to help you understand memory management and threading bugs. (Threading refers to multiple processes of execution -- we haven't covered that yet but you will learn about that later on in your undergraduate careers). First, however, are 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.

**Exercise:**

Another file that in lab6 directory is ***leak.c***

/\* leak.c - memory leak example

This program will reserve a block of memory large enough

to hold 100 characters

\*/

#include

#include

int main()

{

char \*p = malloc(sizeof(char) \* 100);

p = "Constantionople" ;

printf("%s", p);

return 0;

}

* 1. Compile leak.c using

gcc -Wall -g -o leak leak.c

* 1. Run the program using

valgrind --tool=memcheck --leak-check=yes ./leak

* 1. View the following:
  2. Memcheck, a memory error detector
  3. ==18298== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
  4. ==18298== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
  5. ==18298== Command: ./leak
  6. ==18298==
  7. Constantionople==18298==
  8. ==18298== HEAP SUMMARY:
  9. ==18298== in use at exit: 100 bytes in 1 blocks
  10. ==18298== total heap usage: 2 allocs, 1 frees, 1,124 bytes allocated

==18298==

==18298== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1

==18298== at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)

==18298== by 0x400577: main (in lab06/leak)

==18298==

==18298== LEAK SUMMARY:

==18298== definitely lost: 100 bytes in 1 blocks

==18298== indirectly lost: 0 bytes in 0 blocks

==18298== possibly lost: 0 bytes in 0 blocks

==18298== still reachable: 0 bytes in 0 blocks

==18298== suppressed: 0 bytes in 0 blocks

==18298==

==18298== For counts of detected and suppressed errors, rerun with: -v

==18298== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

The highlighted lines in the center show how much memory is lost and

where in the program the lost memory was allocated.

* 1. Edit *leak.c* to correct the problem, then run Valgrind
  2. again to make sure you corrected all problems. Insert comments in
  3. leak.c, to explain the problem with the program.
  4. There will be times when the *--leak-check=yes* option will not
  5. result in showing you all memory leaks. To find absolutely every
  6. 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

For additional Practice with Valgrind see,

[Valgrind Quick Start](http://valgrind.org/docs/manual/quick-start.html)   
[Valgrind tutorial](http://www.cprogramming.com/debugging/valgrind.html)

**Submission**

Use [handin](https://handin.cs.clemson.edu/) to submit *answers.txt, broken2.c,*and *leak.c* files.