CS130 - LAB - Debugging

Name: Anand Mahadevan SID: 862132182

Today's lab will be about debugging programs using GDB and valgrind. If you are using Linux/MacOS, GDB should be already installed. You can install valgrind on Ubuntu using $ sudo apt get install valgrind. If you are using a Windows or Mac machine, you will need to use of the machines with Linux in the department to finish this lab. You can connect remotely to the machines in the lab if you want to test this at home.

1 GDB

GDB helps you understand the execution of the program by allowing you to run a code line by line and check variable values on-the-fly. Say we have the following program called factorial in a file `factorial.cpp`:

```
float factorial(int n)
{
  float i = n;
  for(n--; n >= 0; --n)
    i *= n;
  return i;
}
```

To run GDB or valgrind, we need to compile factorial with debug symbols and we can do this by passing `-g` to the GCC compiler. To run factorial with GDB, we type `gdb factorial`. This will start GDB and load the debug symbols. We can run the program by typing `run`. If the program crashes, it will stop at the part of the code where the problem happened. To see the code where the problem happened, you can type `list`. We can also see what the value of the variables are by typing `print <variable>`. For instance, if we want to see the value of `n` in line 4 of factorial, we can type `print n`.

Before you run the program (while in gdb), you can also add breakpoints that will make the program stop at a specific line of code before continuing. To do this, you can type `breakpoint <filename>:<line of code>`. For instance, if we want to check the values `n` in `factorial.cpp`, we can type `breakpoint factorial.cpp:4`.

A quick guide to GDB can be found at:
`https://web.eecs.umich.edu/~sugih/pointers/summary.html`

Valgrind

Valgrind helps us understand if there are memory violations in our program (among other things). For instance, the following program may not crash but we know it is wrong because we should not be accessing a memory position at index 2 of the array.

```
int main()
{
  int *array = new int[2];
  array[0] = 0;
  array[1] = 0;
  array[2] = 0; // what will happen here?
  return array[2];
}
```

We can run valgrind by typing `valgrind <program call>`. Assuming the above code binary is called test, then we can do `valgrind test`. Here is the output of valgrind when we run test:

```
==16319== Memcheck, a memory error detector
==16319== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==16319== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==16319== Command: ./test
==16319==
==16319== Invalid write of size 4
==16319==    at 0x1086B0: main (main.cpp:5)
==16319==  Address 0x5b82c88 is 0 bytes after a block of size 8 alloc'd
==16319==    at 0x4C3089F: operator new[](unsigned long) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==16319==    by 0x10868B: main (main.cpp:2)
==16319==
==16319== Invalid read of size 4
==16319==    at 0x1086BA: main (main.cpp:6)
==16319==  Address 0x5b82c88 is 0 bytes after a block of size 8 alloc'd
==16319==    at 0x4C3089F: operator new[](unsigned long) (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==16319==    by 0x10868B: main (main.cpp:2)
==16319==
==16319==
==16319== HEAP SUMMARY:
==16319==     in use at exit: 8 bytes in 1 blocks
==16319==   total heap usage: 2 allocs, 1 frees, 72,712 bytes allocated
==16319==
==16319== LEAK SUMMARY:
==16319==    definitely lost: 8 bytes in 1 blocks
==16319==    indirectly lost: 0 bytes in 0 blocks
==16319==      possibly lost: 0 bytes in 0 blocks
==16319==    still reachable: 0 bytes in 0 blocks
==16319==         suppressed: 0 bytes in 0 blocks
==16319== Rerun with --leak-check=full to see details of leaked memory
==16319==
==16319== For counts of detected and suppressed errors, rerun with: -v
==16319== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

The first problem is an invalid write of size 4 (bytes) on line 5. The second is a read of the same memory address. Note also the “definitely lost” line which is saying that we allocated 8 bytes but we never freed that memory, which is also a problem. The code we will be working with can be found on iLearn.

1. Run valgrind on `prog-1`.

   (a) What type of error do we get and why?

   Invalid read of array because the first for loop in main() is out of bounds accessing elements in array a, and lost memory in the resize method where old data memory isn’t being freed.

   (b) How can `prog-1` be changed so we don’t get this error anymore? Note that a correct solution will allow the loop to run to 1000000 in a reasonable amount of time.

   The first for loop in main can be changed to have an if statement to ensure that the element exists in array a before accessing it. And a delete data before replacing data with new data to prevent lost memory.

   Continue these two steps and update the appropriate section until there are no more errors found.

2. Run valgrind on `prog-2`.

   (a) What type of error do we get and why?

   Invalid read of size 4 because of typing issues with object in the set names method, and Access not within mapped region at address because of that same reason.

(b) How `prog-2` can be changed so we don't get this error anymore?

Modify the set names method to have two of them, 1 for sphere, and 1 for point.

3. Run gdb on `prog-3`.

   (a) The program should stop with segmentation fault exception. Type list to see the region where the program stopped. In which line of code is the program crashing?

   Line 11

   (b) Use the command `print <statement>` with the variables that are being accessed on the line where the program is crashing. You can use the `up` command to go back a line if needed. Why does the program crash in this case and how we can fix it?

   The program crashes because v.size() is not returning the intended value of 100. It can be solved by changing the type of i from size t to int.

4. Run valgrind on `prog-4`.

   (a) What type of error do we get and why?

   Invalid read of size 1 because of reading outside of scope, and conditional jump because of char array issues.

   (b) How can `prog-4` be changed so we don't get this error anymore?

   The up and lo arrays are changed to type string, and set to the initial size of the string passed in instead of a constant 8.

5. Run gdb on `prog-5` and follow the steps below.

   (a) The program should stop with a segmentation fault exception. In which line of code is the program crashing?

   Line 61

   (b) Why does the program crash in this case and how we can fix it? (You may want to see the list and node structures in the source code for this.)

   It crashes because removing a node that's the head or tail isn't implemented in the current remove method. I can fix that by adding that functionality.

   (c) Compile and run the program again using gdb. The program should crash again. Try using list and print to figure out why the program is crashing and briefly explain your reasoning. What changes need to be made in the code to fix this problem?

   It crashes because node f is being added after node e which is the current tail. I can fix the current add after method by adding tail checking functionality.

(d) Compile and run the program again using valgrind. The program should display an error. Why do we get this error and how we can fix it?

There is an invalid read in the destructor because the head node is being freed before the rest of the nodes. I can fix that by removing nodes from the tail node backwards, and saving a temporary copy of the last node's prev node before deleting the current node.

6. Using gdb and valgrind (use the best for each situation), briefly describe all problems in `prog-6` and propose fixes for each one of the problems.

(a) In the array constructor with size t parameter, data's size should not be using n when it hasn't be initialized yet, change this to size.

(b) array objects' memories are not being freed, delete each array's data member at the end of main.