Extra Lab Exercise

Debugging with GDB and Valgrind

Objectives

- To learn about debugging with GDB
- To debug memory errors and leaks using Valgrind

Admin

This lab is not marked and there is no submission for it. However, we highly recommend that you attempt it so that you can use GDB and Valgrind in your future labs and assignments.

Resources

You may want to consult the following resources:

- GDB Quickstart: Breakpoints and Printing Values
- Breaking, Stepping Over, and Stepping into Functions
- Debugging - GDB Tutorial (another great tutorial)

Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/21T1/labs/extra/debugging/downloads/lab.zip
```

If you're working at home, download `lab.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`sorter.c`	a buggy program with a simple sorting function
`Set.h`	interface definition for a Set ADT
`Set.c`	buggy implementation of the Set ADT using a binary search tree
`testSet.c`	main program for testing the Set ADT

If you run the `make` command, it will build two executables: `sorter` and `testSet`. Both of these programs are buggy. Before you fix the bugs in the programs, make copies of `sorter.c` and `Set.c` as follows:

```
$ cp sorter.c sorter.bad.c
$ cp Set.c Set.bad.c
```

Task 1 - Debugging the Sorter

The aim of the sorter program is to generate a small array containing random numbers, print it, sort the array using bubble sort, and then print the sorted array. It repeats this process five times, generating different random array contents each time.

If the sorter were correct, you would observe something like the following:

```
$ ./sorter
Test #1
Sorting: 83 86 77 15 93 35 86 92 49 21
Sorted : 15 21 35 49 77 83 86 86 92 93
Test #2
Sorting: 62 27 90 59 63 26 40 26 72 36
Sorted : 26 26 27 36 40 59 62 63 72 90
Test #3
Sorting: 11 68 67 29 82 30 62 23 67 35
Sorted : 11 23 29 30 35 62 67 67 68 82
Test #4
Sorting: 29 02 22 58 69 67 93 56 11 42
Sorted : 02 11 22 29 42 56 58 67 69 93
Test #5
Sorting: 29 73 21 19 84 37 98 24 15 70
Sorted : 15 19 21 24 29 37 70 73 84 98
```

Unfortunately, what you actually observe is:

```
$ ./sorter
Test #1
Sorting: 83 86 77 15 93 35 86 92 49 21
Segmentation fault
```

You may get a different set of random numbers to the above, and maybe even a different error message depending on which machine you're working on, but that doesn't affect the exercise. The program should be able to sort any set of random numbers, but clearly there's a problem.

So, what to do...? You may have noticed that when the programs were compiled, they used the **`-g`** flag, which sets them up to be used with **`gdb`**. Run the program under control of `gdb` to find out where it is crashing.

```
$ gdb ./sorter
GNU gdb (Debian 8.2.1-2+b3) 8.2.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./sorter...done.
(gdb) run
Starting program: some-long-path-name-ending-in/sorter
...
Test #1
Sorting: 83 86 77 15 93 35 86 92 49 21

Program received signal SIGSEGV, Segmentation fault.
0xXXX...XXX in sort (a=0xXXX...XXX, n=10) at sorter.c:39
39              if (a[j] < a[j - 1]) {
(gdb)
```

where the *XXX...XXX* are large hexadecimal numbers, which may vary from machine to machine. The exact value is not important.

The `gdb` command can be quite verbose, and part of the skill of using it is working what to ignore. I've highlighted the critical output in red above. If the long header annoys you, simply use the **`-q`** option:

```
$ gdb -q ./sorter
Reading symbols from ./sorter...done.
(gdb)
```

In the sample output above, you can see the line where the error has occured. Sometimes it's useful to get more context than just a single line. You can do this from within `gdb` using the `list` command, e.g.

```
(gdb) list
34  {
35      int i, j, nswaps;
36      for (i = 0; i < n; i++) {
37          nswaps = 0;
38          for (j = n - 1; j > i; j++) {
39              if (a[j] < a[j - 1]) {
40                  int tmp;
41                  tmp = a[j];
42                  a[j] = a[j - 1];
43                  a[j - 1] = tmp;
(gdb)
```

An alternative to using `list` is simply to keep the program open in an edit window while you run `gdb` in a separate window. GDB also provides a mode so that you can monitor the code and do debugging in a single terminal window; run the `gdb` command with the `-tui` option. Yet another alternative is to use a program like `ddd`, which provides a GUI front-end to `gdb`. For this lab, it may be simpler to stick with plain `gdb`, which has the advantage that it will be available on all Linux machines.

Use `gdb` to find out more information about the state of the program at the point where it crashed. You can find out about the current state of your program in `gdb` using commands like `where` and `print`:

```
(gdb) where         // verify where the program was executing when it crashed
...                 // - gdb gave you a line number above; this will tell you which function
(gdb) print n       // show the value of the parameter 'n'
...
(gdb) print a       // show the value of the parameter 'a'
...                 // - this is the address of the start of the array
(gdb) print *a      // show the first element in the array
...
(gdb) print a[0]    // show the first element in the array
...
(gdb) print a[2]    // show the third element in the array
...
(gdb) print *a@5    // show the first 5 elements in the array
...
(gdb) print a[j]    // show the j'th element of the array
...
```

Keep examining variables until you find something that looks anomalous. You will then need to find out how it got that way. You could look at the code again and you might spot the error. If not, continue...

One useful way to find out how your program reached its current erroneous state, is to set a breakpoint on the `sort` function and observe the behaviour as that function executes.

```
(gdb) break sort
...
(gdb) run
...                 // stops at breakpoint ... start of sort function
(gdb) next
...                 // execute next statement, then check variable values
(gdb) next
...
```

If examining the variables at each step doesn't help you to find the problem quickly, then try adding a breakpoint on line 39 (where the error occurs), and re-running the program. After it stops each time, check the value of variables. After each stop/check, you can continue the program with the **`continue`** command.

Once you've found the problem, change the code to try to fix it, recompile, and see whether the program now exhibits the expected behaviour.

Task 2 - Debugging the Set

If you simply compile the `testSet` program without change, it will behave as follows:

```
$ ./testSet
Test 1: Create set
Passed
Test 2: Add to set
Segmentation fault
```

Note that this program uses assertions to aid debugging. While assertions provide some information, they may not provide enough to work out what the problem is (e.g. "what is the value of variable `i`?").

Now run the program under `gdb`'s control, observe the values of variables when it crashes, and use this information to determine the causes of the problems.

Note that **this program has multiple bugs**, so after you fixed one, another will probably manifest itself when you recompile and test. Repeat the above until `testSet` exhibits the expected behaviour.

Once the Set ADT has been implemented correctly, then the `testSet` program should produce something like the following:

```
$ ./testSet
Test 1: Create set
Passed
Test 2: Add to set
Passed
Test 3: Add duplicates
Passed
Test 4: Add more to set
Passed
Test 5: Print set
{2, 4, 6, 7, 9}
Check manually
Test 6: Free set
Now check for memory errors and leaks using valgrind
```

Note that even though a program behaves as expected, this does not guarantee that the code is correct. The code may contain memory errors, which occur when your program tries to read from or write to a memory location that it shouldn't. The code may also contain memory leaks, which occur when your program dynamically allocates memory (using *malloc*), but doesn't free it once it's no longer needed. Memory errors are more difficult to debug, as they don't always manifest themselves, so a program with memory errors may run normally one time, but abnormally the next. They also often lead to strange behaviour that occurs far away from the source of the actual problem.

Run the program in `valgrind` to see if the code contains any memory errors or leaks. If your code *does* contain memory errors or leaks, you might get output that looks like the following:

```
$ valgrind ./testSet
==15336== Memcheck, a memory error detector
==15336== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==15336== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==15336== Command: ./testSet
==15336==
Test 1: Create set
==15336== Invalid write of size 4
==15336==    at 0x1098A4: SetNew (Set.c:38)
==15336==    by 0x1091BD: main (testSet.c:18)
==15336==  Address 0x4a43488 is 0 bytes after a block of size 8 alloc'd
==15336==    at 0x48357BF: malloc (vg_replace_malloc.c:299)
==15336==    by 0x10985F: SetNew (Set.c:31)
==15336==    by 0x1091BD: main (testSet.c:18)
...
...
...
==15336== HEAP SUMMARY:
==15336==     in use at exit: 56 bytes in 3 blocks
==15336==   total heap usage: 7 allocs, 4 frees, 1,152 bytes allocated
==15336==
==15336== LEAK SUMMARY:
==15336==    definitely lost: 56 bytes in 3 blocks
==15336==    indirectly lost: 0 bytes in 0 blocks
==15336==      possibly lost: 0 bytes in 0 blocks
==15336==    still reachable: 0 bytes in 0 blocks
==15336==         suppressed: 0 bytes in 0 blocks
==15336== Rerun with --leak-check=full to see details of leaked memory
==15336==
==15336== For counts of detected and suppressed errors, rerun with: -v
==15336== ERROR SUMMARY: 19 errors from 19 contexts (suppressed: 0 from 0)
```

Valgrind gives detailed information of memory errors. If your program tried to read from an invalid memory location, Valgrind will report an invalid read. If your program tried to write to an invalid memory location, Valgrind will report an invalid write and tell you the size of the data item that your program wrote. In the above example, Valgrind reported an 'Invalid write of size 4', which means the program

likely tried to write an `int` to an invalid memory address (since an `int` is 4 bytes). Valgrind will also tell you the line on which the error occurred. For example, `(Set.c:38)` means Line 38 in `Set.c`.

From this output, it is up to you to figure out the cause of the error and fix it. Here are some common causes of memory errors:

- Not allocating enough memory - this is common with strings
- Trying to access an index beyond the end of an array
- Reading and using an uninitialised value
- Use after free - this is where you free a block memory and then try to access it afterwards.
- Double free - this is where you free the same block of memory twice.

Valgrind also summarises memory leaks. You can get more detailed information about memory leaks by using the `--leak-check=full` option, as the output above suggests.

```
$ valgrind --leak-check=full ./testSet
...
...
...
==15280==
==15280== HEAP SUMMARY:
==15280==     in use at exit: 56 bytes in 3 blocks
==15280==   total heap usage: 7 allocs, 4 frees, 1,152 bytes allocated
==15280==
==15280== 8 bytes in 1 blocks are definitely lost in loss record 1 of 3
==15280==    at 0x48357BF: malloc (vg_replace_malloc.c:299)
==15280==    by 0x10985F: SetNew (Set.c:31)
==15280==    by 0x1091BD: main (testSet.c:18)
==15280==
==15280== 24 bytes in 1 blocks are definitely lost in loss record 2 of 3
==15280==    at 0x48357BF: malloc (vg_replace_malloc.c:299)
==15280==    by 0x1099F7: newNode (Set.c:80)
==15280==    by 0x109982: doSetAdd (Set.c:68)
==15280==    by 0x1099D1: doSetAdd (Set.c:74)
==15280==    by 0x1099D1: doSetAdd (Set.c:74)
==15280==    by 0x109944: SetAdd (Set.c:61)
==15280==    by 0x1095B2: main (testSet.c:66)
==15280==
==15280== 24 bytes in 1 blocks are definitely lost in loss record 3 of 3
==15280==    at 0x48357BF: malloc (vg_replace_malloc.c:299)
==15280==    by 0x1099F7: newNode (Set.c:80)
==15280==    by 0x109982: doSetAdd (Set.c:68)
==15280==    by 0x1099A4: doSetAdd (Set.c:72)
==15280==    by 0x1099D1: doSetAdd (Set.c:74)
==15280==    by 0x109944: SetAdd (Set.c:61)
==15280==    by 0x1095C3: main (testSet.c:67)
==15280==
==15280== LEAK SUMMARY:
==15280==    definitely lost: 56 bytes in 3 blocks
==15280==    indirectly lost: 0 bytes in 0 blocks
==15280==      possibly lost: 0 bytes in 0 blocks
==15280==    still reachable: 0 bytes in 0 blocks
==15280==         suppressed: 0 bytes in 0 blocks
==15280==
==15280== For counts of detected and suppressed errors, rerun with: -v
==15280== ERROR SUMMARY: 22 errors from 22 contexts (suppressed: 0 from 0)
```

Valgrind will tell you where the memory that was leaked was allocated. From there, you should be able to figure out why you aren't freeing the memory and where you should be freeing it.

Once you have fixed all the memory errors and leaks, Valgrind should output something like the following:

```
$ valgrind ./testSet
==22805== Memcheck, a memory error detector
==22805== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==22805== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22805== Command: ./testSet
==22805==
Test 1: Create set
Passed
Test 2: Add to set
Passed
Test 3: Add duplicates
Passed
Test 4: Add more to set
Passed
Test 5: Print set
{2, 4, 6, 7, 9}
Check manually
Test 6: Free set
Now check for memory errors and leaks using valgrind
==22805==
==22805== HEAP SUMMARY:
==22805==     in use at exit: 0 bytes in 0 blocks
==22805==   total heap usage: 7 allocs, 7 frees, 1,160 bytes allocated
==22805==
==22805== All heap blocks were freed -- no leaks are possible
==22805==
==22805== For counts of detected and suppressed errors, rerun with: -v
==22805== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```
