Week 2 Lab Exercise

Debugging with GDB

Objectives

- to practice debugging with GDB

Admin

Marks	5=outstanding, 4=very good, 3=adequate, 2=sub-standard, 1=hopeless
Submit	`give cs2521 lab02 sorter.c Queue.c` or via WebCMS
Assessment	The marks for this lab are based on you demonstrating to your tutor that you can use `gdb` to debug the two supplied buggy programs. You should also submit the now bug-free programs.
Deadline	submit by 11:59pm Sunday of Week 2

Online resources

You may want to consult the following resources: **GDB: online resources**

Setting Up

Set up a directory for this lab under your `cs2521/labs` directory, change into that directory, and run the following command:

```
$ unzip /web/cs2521/20T2/labs/week02/lab.zip
```

If you're working at home, **download from** here.

You should now have the following files in your directory:

`Makefile`	a set of dependencies used to control compilation
`Queue.h`	interface definition for a FIFO Queue ADT
`Queue.c`	buggy implementation for the Queue ADT
`testQ.c`	main program for testing the Queue ADT
`sorter.c`	buggy program with a simple sorting function
`unsort.c`	skeleton program for the Extra Challenge part

If you run the `make` command, it will build two executables: `testQ` and `sorter`. Both of these programs are buggy.

Before you fix the bugs in the programs, make copies of `sorter.c` and `Queue.c` as follows:

```
$ cp sorter.c sorter.c.bad
$ cp Queue.c Queue.c.bad
```

You should restore these bad versions when you want to demonstrate your prowess with the debugger to your tutor.

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 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 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
BFD: /usr/lib/debug/.build-id/f2/5dfd7b95be4ba386fd71080accae8c0732b711.debug: unable to initialize decompress st
...
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 absolute value is not important.

Also, the lines that begin with `BFD: /usr/lib/debug/...` may not appear; they are temporary glitches in our gdb setup. If they do appear, ignore them.

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 **`cont`** 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.

Debugging the Queue

If you simply compile the `testQ` prograM without change, it will behave as follows:

```
$  ./testQ
Test 1: Create queues
Passed
Test 2: Add to queues
testQ: testQ.c:31: main: Assertion `queueLength (q2) == i' failed.
Aborted
```

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

Once the queue ADT had been implemented correctly, then the `testQ` program should produce something like the following:

```
$ ./testQ
Test 1: Create queues
Passed
Test 2: Add to queues
Final q1: H 01 02 03 04 05 06 07 08 09 10 T
Final q2: H 01 02 03 04 05 06 07 08 09 10 T
Passed
Test 3: Remove from queues
Passed
Test 4: Destroy queues
Passed
Test 5: Remove from emoty queue
This test should fail an assertion
testQ: Queue.c:77: leaveQueue: Assertion `q->size > 0' failed.
Aborted
```

Note that it fails the final test. This test aims to check removing an item from an empty queue, which, of course, should fail. There are ways of catching such an error without simply aborting. (If you wish, you could try to implement a better `testQ` which actually says "Test Passed" if the queue ADT recognises the error of trying to remove from an empty queue and continues on. (But only do this after you've completed this exercise.)

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

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 `testQ` exhibits the expected behaviour.

Demonstration

Save copies of your fixed versions of `sorter.c` and `Queue.c` and replace them with the bad versions that you saved earlier (did you?). Then rebuild your executables as follows:

```
$ mv sorter.c sorter.c.good
$ mv sorter.c.bad sorter.c
$ mv Queue.c Queue.c.good
$ mv Queue.c.bad Queue.c
$ make clean
$ make
```

Now show your tutor how you were able to use `gdb` to debug each of the programs, and they will note that you have completed the practical part of this lab. Note that **both partners** in each lab pair are expected to demonstrate something; one person should demonstrate debugging on the sorting program, while the other person should demonstrate debugging on the Queue ADT. Note also that the demonstration is not just saying "Here are the bugs I found"; the most important part of the demo is showing **the process** that you used to find them.

Extra Challenge - Unsorting

The Linux `sort` command has various manifestations, some of which provide a **`-R`** option to shuffle the elements of the array (**R**andomise the order). If we were working on a system with a `sort` command that did not have the ability to take in a sorted sequence and randomly re-order it, we could write a program to do this. There is a skeleton for just such a program in the file **`unsort.c`**. This program either reads its input from a named file, or takes it from standard input. It should interpret its input as a sequence of numbers and write a randomly re-ordered sequence of these numbers, one per line, to its standard output.

For this task, complete the `unsort` program. It should work even for inputs with very large numbers of values. See if you can make it faster than the Linux **`sort -R`** command, even for sequences of more than 100000 numbers.

As an example of how the program works, consider that we have a file called `nums` containing the numbers from 1 to 5 in ascending order. The following shows *possible* usage of this program:

```
$ ./unsort nums
2
5
3
1
4
$ seq 1 9 | ./unsort
3
7
2
```

```
5
6
4
9
1
8
$
```

The precise order of numbers in the output does not matter, and does not have to match the output above. It simply has to look "random enough". (How could you determine whether it is "random enough"?)

There are no marks for doing this ... just the satisfaction of writing something that's better than a Linux command, at least for this one task.

Have fun, *jas*
