COMP26120

Academic Session: 2018-19

Lab Exercise 3: Debugging; Arrays and Memory Management

Duration: 1 lab session

For this lab exercise you should do all your work in your COMP26120/ex3 directory. Copy the starting files from */opt/info/courses/COMP26120/problems/ex3*.

Important: Read the supporting material on (i) pointers and arrays, and (ii) how to debug C programs before the lab starts.

Learning Objectives

At the end of this lab you should be able to:

1. Identify some common mistakes related to strings and memory management in C;
2. Use structs, unions, pointers, malloc and free in C;
3. Explain some fundamental differences between C and Java - the explicit use of pointers and of memory management;
4. Use GDB and valgrind to help debug C programs.

Introduction

We **strongly** recommend you review the lecture slides, and the two sets of reading material on *pointers and arrays* and *debugging* before starting this lab. See the course unit website for these documents.

You might also find The GNU C Programming Tutorial helpful now and throughout this course. The relevant parts to this lab are Pointers, Strings, and Data Structures.

Part 1

In this part you will take the provided C program *broken.c* and produce a C program *fixed.c* and a text file *part1.txt* describing what you did.

You do not need to get to the end of this part to get full marks. You need to have made some reasonable progress and write down what you did. You will also need to demonstrate how to use a debugger and Valgrind to the TA and explain what they do.

This program (*fixed*) should take a single string as input on the command line and print the result of making the first letter of each word upper-case and the rest of the letters lower-case. The output should first print the input and then the result in the following way:

```
./fixed "hELLO wORLD"
hELLO wORLD becomes Hello World
```

Your fixed file will be tested with respect to this specification. Note that non-alphabetic symbols should be preserved and a word is defined as that appearing at the start of the string or directly after a whitespace character.

You should use a debugger and memory checker to debug *broken.c*. The linked reading material[1] gives further details and there is a lot of help to be found online.

Here we give advice on how to do this with GDB and Valgrind but you can use other tools as long as you can explain them. We've provided a script to help but if you make different fixes you will get diffrent results. THIS DOES NOT MATTER.

To start with GDB we suggest running:

```
make broken
gdb broken
...
(gdb) run "hELLO wORLD"
```

You should see something like

```
Program received signal SIGSEGV, Segmentation fault.
0x0000003973298bf6 in __strcpy_sse2_unaligned () from /lib64/libc.so.6
```

Now run the command

```
(gdb) backtrace
```

to see the current call stack. This will tell us that there was a *Segmentation fault* inside strcpy. What did we forget to do with `work.second`? **Hint:** what do pointers point to? What does `malloc` do?

Make a note of what the issue was and how you fixed it in part1.txt.

Once you have fixed the segmentation fault repeat

```
make broken
gdb broken
...
(gdb) run "hELLO wORLD"
```

and you might see something like

```
hello world becomes (null)
```

is this what you expected? As noted above we should have seen

```
hELLO wORLD becomes Hello World
```

There are two things wrong with our output. Firstly, we see `(null)` where we wanted `Hello World` and secondly the input is printed as `hello world` instead of `hELLO wORLD`. You need to explain and fix both of these problems.

[1] Note that here we use the command line `gdb` whereas the reading material recommends `ddd`. You may use either.

For the first issue try running

```
gdb broken
(gdb) break process
(gdb) run "hELLO wORLD"
(gdb) watch work.first[1]
```

This first sets a breakpoint to pause execution when the `process` function starts. We then run the program and it will pause where we asked it to. We then ask it to watch the second character in the `work.first` string, as this is the first character that is changed. The result should tell you where this string is being changed and hopefully help you find the bug. **Hint:** if you are still unsure look at the concept of aliasing.

For the second issue set a breakpoint for after memory is allocated for `work.second` (you fixed that issue above didn't you...) and then run

```
(gdb) info args
work = {first = 0x7fffffffc8a3 "hello world", second = 0x602010 ""}
(gdb) up
(gdb) info locals
work = {first = 0x7fffffffc8a3 "hello world", second = 0x0}
```

Your output is likely to differ in terms of pointer addresses! What did we just do? The command `info args` prints the argument to `process` where we can see `work.second` pointing to some memory. The command `up` makes us inspect the stack frame above us (the one in `main`) and the command `info locals` shows the local variables in that stack frame. In that version of `work` the value of `second` is still null. The two versions of `work` are different; when we called `process` we *copied* `work`. **Hint:** what does it mean to pass by value? what is a reference?

You should now be able to fix those two issues in some way.

Make a note of what the two issues were and how you fixed them in part1.txt.

Have you now spent a long time on this part? Remember you do not need to get to the end, but you should at least try running valgrind.

Once they are fixed I would expect you to have something that does the following:

```
./broken "hELLO wORLD"
hELLO wORLD becomes hEllo World
```

which has wrongly capitalised the second character of `Hello`. To fix this bug you may wish to use `gdb` to investigate where `ptr` is pointing to at different points in the loop.

Make a note of what the issue was and how you fixed it in part1.txt.

Once the output looks right you are almost there. Now run

```
valgrind ./broken ‘‘hELLO wORLD’’
```

and you will almost definitely see some complaints unless you remembered to add appropriate `free` commands earlier *and* remembered that strings need a null-terminator *and* noticed the logical bug in the first loop that reads from a bad bit of memory. Once Valgrind stops complaining you can rename `broken.c` to `fixed.c` and you are done.

Make a note of what the issues were and how you fixed them in part1.txt.

First Note: you *must* fix this code rather than starting from scratch and you will be asked to explain, and possibly demonstrate, how you have used a debugger and memory checker in the process.

Second Note: you don't need to get your code passing all of the tests. Notice that most of the marks here are for being able to explain how to use the tools. Therefore, you should limit the amount of time you spend on this part if you are getting stuck.

Part 2

In this part you are given a pair of programs *arrays.h* and *arrays.c*. In this part you should modify *arrays.c* **but not** *arrays.h* (as we will swap this out for alternative header files during testing). You should probably keep a working back-up each time you progress to a new step.

There is every chance that at some point your program will crash with the message: *Segmentation fault*. You may need to use your new found debugging skills (from Part 1) to work out what happened.

You will continue using your program in lab exercise 4, so make sure you leave it in a good state.

Step A: Array of Struct

Edit the program to:

- Declare a *struct* (type) that describes a person. It should contain a string for a name and an int for an age (in years).
- Declare an array of these structs (called *people*).
- Complete the *insert* function to insert a name and age into the next unused element in the arrray. Use a *static* variable[2] (e.g. called *nextinsert*) inside the function to remember where the next unused element is (this is not good practice but you will get rid of it later).
- Inside *main*:

  In a *for* loop, call *insert* to put the next name and age into the *people* array.

  Use a **second** loop to print the contents of the *people* array.

It is important that you pass the complete array of structs to *insert*, not just one element of it.

Step B: Array of Pointer to Struct

Edit your program so that your array *people* is now an array of pointers to structs. Modify *insert* to call *malloc* to create a new struct and set the correct array element pointing to it. Remember to check the result of *malloc* for errors (and you can test this by temporarily giving *malloc* a ridiculously big number as a parameter).

[2] These are a bit different from the notion of `static` in Java. Take a look here (yes, I did just link to StackOverflow).

Step C: Tidying up using *free* and *valgrind*

Edit your program so that, after the array has been printed in *main*, you use a **third** loop to call *free* to release the memory allocated by *malloc*.

Use valgrind to make sure you have got this part right (there are no memory leaks). You should already have used this in Part 1. In case you didn't you can find more information in the related reading and by running *man valgrind*.

Step D: A Pointer (or "ref" or "var") Parameter

Finally we're going to get rid of the static *nextinsert* variable.

- Move the declaration of *nextinsert* from inside *insert* to inside *main* (i.e. following the declaration of your array *people*).

  Get rid of the *static* modifier on the declaration, but keep the initialisation to 0.

- Add *nextinsert* as a parameter to the declaration of *insert* (i.e. `static void insert (..., int nextinsert))` and to the call from main (i.e. `insert (..., nextinsert);`).

Compile and run your program as normal. It should fail, with only one person in your array at index 0, and probably a "Segmentation fault" as it hasn't set up the other array items (use gdb to find out which line your program crashed at).

The problem is that, any parameter inside a function is only a local copy of the value passed into the function. Although *insert* increments the parameter *nextinsert*, the changed value doesn't get returned from the function at the end of the call, so *nextinsert* in *main* never changes and each new person is being put into the first array item (i.e. at [0]).

To fix this, we could try to use the return result from the function, but instead we are going to make the parameter a **reference** (i.e. **pointer**) to the variable whose value we want to modify. This is similar to the way *sscanf* puts input values into variables e.g.:`sscanf(argv [2], ''%f'', &float_value);`

In this way, the **address** of nextinsert is passed into the function, then *insert* **dereferences** the pointer parameter and increments the value.

- Change the **declaration** of `insert` to use a pointer parameter:

  `..., int *nextinsert)`

- Change the **call** of `insert` to use a pointer parameter:

  `..., &nextinsert)`

- Change each **use** of `nextinsert` inside `insert` to access the integer value via the pointer:

  `...* nextinsert ...`

  You may need to use brackets e.g. `(*nextinsert)++`

Your program should now behave properly again.

Note:

Declaring *nextinsert* as an **integer** and passing the address of it to the **call** of *insert* using an address-of operator &:

`..., &nextinsert)`

would be the same as declaring *nextinsert* as a **pointer** (Read the manual of how to declare and initialize a pointer), then directly pass it to the **call** of *insert*.

```
..., nextinsert)
```

Marking Scheme

You will continue using your program in lab exercise 4, so make sure you leave it in a good state.

Has the student made an attempt to fix the bugs in broken.c?	
More than the first bug was fixed	(1)
The first bug was fixed	(0.5)
No bugs were fixed	(0)

Can the student explain what they did?	
The student has written an understandable account of how they attempted to fix bugs in broken.c and can demonstrate how to use a debugger and memory checker (e.g. gdb and valgrind). Importantly, they can explain what the bugs they fixed were and how their fixes work.	(2)
The student has written something brief but can explain everying and run the tools.	(1.5)
The student has written something but it is not that easy to understand. They can explain their fixes but some details might be missing. They can run the tools.	(1)
The student made an attempt at explaining what they did.	(0.5)
The student did not attempt this part.	(0)

Part 2 is complete	
All parts have been completed and the code is of a reasonable quality e.g. good variable names, suitable comments, and well-structured.	(3)
All parts have been completed but the code quality is poor	(2.5)
Steps A-C are complete but D is missing	(2)
Steps A and B are complete	(1)
No attempt has been made	(0)

Part 2 is correct	
All the tests pass.	(2)
All but the last test (testing malloc returning NULL) pass.	(1)
The output is wrong or valgrind detects a memory leak.	(0)

Understanding	
The student can explain the key issues around memory management in C, including the copy mechanism for function calls	(2)
There are some gaps in understanding but most questions are answered correctly	(1)
The student can answer some questions but needs to revise this topic	(0.5)
The student cannot answer any questions	(0)