CPSC 213 – Assignment 5

Memory Management and Structs

Due: Saturday, February 9, 2019 at 11:59pm
After an 8-hour grace period, no late assignments accepted.

Learning Objectives

Here are the learning objectives of this assignment, which will be examined in this week's quiz. They are a subset of the unit learning objectives listed on Slide 2 of Unit 1c.

After completing this assignment you should be able to:

1. identify and correct dangling-pointer and memory leak bugs in C caused by improper use of free();
2. write C code that uses techniques that avoid dynamic allocation as a way to minimize memory-allocation bugs; and
3. write C code that uses reference counting as a way to minimize memory-allocation bugs.

Goal

In this week's assignment you will do two things. First, you get additional practice reading assembly code, figuring out what C program it came from, and counting memory references — skills that will be useful on the midterm and beyond.

Second, will examine dynamic allocation and de-allocation in C. The goal here is to help you clarify your understanding of this topic, particularly dangling pointers, memory leaks, and how to avoid them. You will do this by examining two programs that contain dangling pointer bugs to identify the bugs and fix them.

Download the code file

Download the file *www.ugrad.cs.ubc.ca/~cs213/cur/assignments/a5/code.zip*. It contains the following files, some in subdirectories named `q2`, `q3`, and `q4`:

1.	`q1.[cs]`	– Question 1
2.	`refcount.[ch]`	– Questions 2-4
3.	`q2`: `Makefile`, `q2stack.c`	– Question 2

4. `q3`: `integer.[ch]`, `helper.[ch]`, `main.c`, `Makefile`, `set.[ch]`, `stack.[ch]` – Question 3
5. `q4`: `list.[ch]`, `tree.[ch]`, and `main.c, Makefile` – Question 4

Question 1 — Reading Assembly Code [10%]

Read the included `q1.c` and `q1.s` files. You will see that `q1.c` declares two structs, allocates some objects from those structs and initializes them. It has a procedure named `q1()` that is blank. The code listed in `q1.s` implements the code in this procedure (without the procedure call itself). Read `q1.s` carefully and run it through the simulator to figure out how it manipulates these structs. Then do the following.

1. [2%] Comment every code line of `q1.s` with a high-level comment (i.e, a C-like comment).
2. [6%] Modify the procedure `q1()` in `q1.c` to perform the same computation as `q1.s`. Do not modify the other parts of this C file; they are used to test and mark your code. Note that you can optionally provide different values for the structs on the command line. The C code you write must work for arbitrary values.
3. [2%] Examining the code you write for `q1()` in `q1`. Count the *minimum* number of memory reads and writes that are required to execute these five lines. Note these numbers may be different from `q1.s`, which takes each line individually (e.g., reading the value of `i` each time, which is not really necessary). Ignore register allocation for this questions; i.e., you can assume that you have an infinite number of registers available.

   Place these two numbers in the file `q1.txt`; on two separate lines with the number of reads listed first (just these numbers and nothing else). Then carefully explain your answer by listing each of the reads and writes that are required (give line number and describe the access using variable names) in the file `q1-desc.txt`.

Question 2 – Dynamic Allocation (Part 1) [20%]

The file `stack.c` contains a program that implements a stack data structure and tests it. A stack has two operations: `push` and `pop`. `Push` adds strings to the stack and `pop` removes them, in last-in-first-out order. So, if you `push` “`one`”, “`two`”, and then “`three`”, in that order, three `pops` will give you “`three`”, “`two`”, “`one`”, in that order.

You can use the included `Makefile` to build the program by typing

```
make q2stack
```

This will produce an executable file named `stack` that you can run by typing:

```
./q2stack
```

This program tests the stack by pushing two elements, popping one, pushing two more, popping all three, and then printing the four popped values in the order they were popped. The values pushed are "A", "B", "C", and "D", in this order and so the output should be:

```
B D C A
```

But, as you will see the program does not produce this output. It has a bug.

Find the bug and write a careful description of it in the file `q2.txt`. The bug is a dangling pointer somewhere. If you can't find it, you might try commenting out calls to `free()`. This should fix the dangling pointer bug by replacing it with a memory leak.

Once you have found and described the bug, you need to fix it. Update the file `stack.c` so that it has neither dangling pointers nor memory leaks. Your fix should change the existing program as minimally as possible. In particularly, the program must still call `malloc` where it does and use this dynamically allocated memory as it does.

Test your program by running it to verify that you get the correct output. If you do, this is a good indication that you have fixed the dangling pointer but. But, you are not done. You must also ensure that your program does not have a memory leak. To do so, you must run the program through `valgrind`, which is available on the undergrad servers but likely not on other platforms you might use for other parts of the assignment. The test you need to run is the following:

```
valgrind --tool=memcheck --leak-check=yes ./q2stack
```

If you have a memory leak you will see an error that looks something like this:

```
==31419== LEAK SUMMARY:
==31419==    definitely lost: 240 bytes in 2 blocks
==31419==    indirectly lost: 240 bytes in 2 blocks
```

If you have a dangling pointer you may see an error that looks something like this:

```
==31993== Invalid read of size 1
==31993==    at 0x4E7D000: vfprintf (vfprintf.c:1629)
```

If you eliminate both bugs (your goal) then you should see output something like this:

```
==1272== HEAP SUMMARY:
==1272==     in use at exit: 0 bytes in 0 blocks
==1272==   total heap usage: 4 allocs, 4 frees, 480 bytes allocated
==1272==
==1272== All heap blocks were freed -- no leaks are possible
==1272==
==1272== For counts of detected and suppressed errors, rerun with: -v
==1272== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Or you might see (depending on which version of valgrind you use):

```
==9402== LEAK SUMMARY:
==9402==    definitely lost: 0 bytes in 0 blocks
==9402==    indirectly lost: 0 bytes in 0 blocks
==9402==      possibly lost: 0 bytes in 0 blocks
==9402==    still reachable: 4,096 bytes in 1 blocks
==9402==         suppressed: 25,084 bytes in 373 blocks
```

There are a few different ways to fix the bug. Any of them are fine as long as you still allocate stack elements dynamically using `malloc`.

Question 3 – Dynamic Allocation (Part 2) [30%]

Question 3 is a program that accepts a list of numbers on the command line and adds them to a hash set and to a stack. It then prints the content of the set, pops and prints every element in the stack, and then prints the set again before terminating. This program has a memory leak.

Fix the leaks without adding dangling pointers using reference counting (the Makefile is already setup to use them; add "`#include "refcount.h"`" to C files as needed). You must use the provided `refcount.c` and `.h` files for reference counting. You must not change any of the header files (i.e., files ending in `.h`), you must not add any coupling other than reference counting, and, **you must not change `main.c` in any way.**

Describe the leak and your solution in the file `q3.txt`.

Question 4 – Dynamic Allocation (Part 3) [40%]

Now we turn a similar bug in a more challenging example. This example is a program that consists of three files: `main.c`, `list.c`, and `tree.c`, built by typing `make`. The first file contains `main`, the second implements a doubly-linked list, and the third implements a binary search tree. The implementation is free of all bugs expect for a single dangling pointer bug. Again, your goal is to find the bug, describe it, and fix it.

The program reads a list of strings from the command line, adds them to the linked list in input order, and adds some of them (chosen randomly) to the binary search tree that organizes them in alphabetical order. This tree is an index on the linked list and so tree nodes store pointers to linked-list elements.

The point of this setup is to create a complex memory-management problem in which some list elements have two references (the `prev` and `next` pointers of the linked list) and some have three (the two linked list pointers and a third pointer from the tree). The problem is to correctly reclaim memory allocated to a list element (i.e., by calling `free`) when neither the list nor the tree references it. Doing so is not trivial, because the linked list implementation does not know which of its elements are referenced by the tree.

After creating the list and tree, the test program traverses the tree, arriving at the elements it indexes in alphabetical order, and then using the linked list to print the input list starting with this element.

For example if you ran the program with the input "`one two three four five`", like this:

```
./main one two three four five
```

And if the tree indexed the nodes "`one`", "`three`" and "`five`", it would output:

```
five
one two three four five
```

```
three four five
```

The program then deletes the linked list and every element that is not referenced by the tree and performs the same traversal again. In this step it should produce the output:

```
five
one
three
```

But, you will see that it doesn't, because deleting the list introduces a dangling pointer bug.

Find the bug. As a starting point, you might want to once again comment out all calls to `free` thus removing the dangling pointer bug and allowing the program to run correctly. Of course, doing this introduces a memory leak.

Once you've identified the line(s) of code that cause the bug, carefully describe the bug by placing your description in the file `q4.txt`.

Now, fix the bug using reference counting. You must use the provided `refcount.c` and `.h` files for reference counting (the Makefile is already setup to use them; add "`#include "refcount.h"`" to C files as needed). You must not remove any calls to `malloc` other than by replacing them with calls to `rc_malloc`. You must maintain the modularity of the program. For example, you can not change any of the `.h` files and there must be no coupling between the tree and list files other than for reference counting. Your fix should change the program as minimally as possible.

Modify whichever of the three files you need to and test your program to ensure that it produces the correct output and that `valgrind` reports no memory leaks, just as you did for Step 1.

Note that `valgrind` is limited to checking a single execution of the program, the one you specify on the command line. It is often the case that memory-leak bugs only cause memory leaks for certain inputs. So, be sure to run `valgrind` for a variety of inputs, as we will when we test your program. If *any* execution reports a memory leak, then you have a bug.

What to Hand In

Use the `handin` program.

The assignment directory is `~/cs213/`**`a5`**, it should contain the following ***plain-text*** files.

1. `PARTNER.txt` containing your partner's CS login id and nothing else (i.e., the 4- or 5- digit id in the form a0z1). Your partner should not submit anything.
2. The files `q1.s`, `q1.c`, `q1.txt`, and `q1-desc.txt` for Question 1;
3. subdirectories named `q2`, `q3`, and `q4` just like you see in the code file;
4. in the `q2` directory: `q2.txt` and `q2stack.c`;
5. in the `q3` directory: `q3.txt`, `integer.c`, `helper.c`, `set.c`, and `stack.c`; and
6. in the `q4` directory: `q4.txt`, `list.c`, `main.c`, and `tree.c`.