

CPSC 213 – Assignment 5

Memory Management

Due: Friday, July 19, 2019 at 11:59pm

You may use one of your two late days on this assignment to make it due Saturday.

Learning Objectives

Here are the learning objectives of this assignment, which will be examined in the corresponding 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, we 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'll start by implementing a simple memory allocator, to get you more familiar with how it works under the hood. Then, you will examine a program that contains a dangling pointer bug to identify the problem and fix it.

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` and `q3`:

- | | |
|-------------------------------|------------------|
| 1. <code>q1.[cs]</code> | – Question 1 |
| 2. <code>refcount.[ch]</code> | – Questions 2, 3 |

- 3. `q2: main.c, mymalloc.[ch], Makefile` – Question 2
- 4. `q3: {list,tree,element}.[ch], main.c, and Makefile` – Question 3

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%] Examine the code you wrote for `q1()`. Count the *minimum* number of memory reads and writes that are required to execute these five statements together. 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 — malloc and free [50%]

In this assignment, you will implement a functioning version of `malloc` and `free` from scratch. In class, I showed you a *real* malloc implementation that runs to over 5000 lines. Obviously, we won't be implementing anything near that level of complexity!

The goal is to arrive at a simple, yet functional implementation which is able to service requests using an *explicit free list* (textbook section 9.9.13; basically, freed blocks are added into a linked list and searched during malloc). You will be guided through the approach, but start early because it's easy to make mistakes and have to spend time debugging.

To start off, look at the code sample provided. It has a Makefile and two main modules: `mymalloc.c`, `{c,h}` and `main.c`. You will modify and hand in the implementation in `mymalloc.c`, but **leave `mymalloc.h` and `main.c` alone** (you can modify them for local testing if you like, but the autograder will not use your versions). You can add helper functions, variables, and constant as you wish in `mymalloc.c`.

`mymalloc` exports three functions. Read through the documentation in the header file to know what they do, and see `main.c` for how they get used. `mymalloc.c` also contains a very simple implementation of `malloc` (adapted from the three-line version shown in class) which does not support `free` at all.

`main.c` implements three test-cases which will stress your `malloc` implementation. Your goal is to pass all of the tests. The autograder will use the exact same tests, so if you pass every test locally you should be able to pass the tests on the autograder. To compile, run `make` in this directory to make the `mymalloc` binary.

To run a test, run `./mymalloc <test name>`, or use `./mymalloc all` to run every test in sequence. The provided, dumb implementation of `malloc` passes some tests, but fails most of them.

Implementing `malloc` and `free`

1. First, modify your `malloc` implementation to allocate a little extra memory, and return an offset pointer, similar to what `realloc.c` does. In the extra space, you will store some *metadata* about the memory block. As mentioned in class, the problem with the dumb implementation is that it doesn't know the *size* of the allocated blocks; therefore, you will store the allocated size in the metadata area for `free` to use. Test your implementation to make sure it still passes the same tests as before.
2. Next, you'll implement `free`. Remember that when `free` is called, the user is saying that the memory is no longer in use, so the allocator (`malloc/free`) can use the memory as it wishes. You're going to build a *singly-linked list* out of all the blocks of free memory. Create a static *head* pointer to point to the most recently freed chunk of memory. When a new chunk is freed, write the address of the old *head* into the freed memory, thus forming a linked list out of the freed chunks. *Note: You'll need to make sure your allocated chunks are big enough for a pointer - which will be 8 bytes in size on the autograding machine!*
3. Finally, you'll augment `malloc` to walk over the free list. It will have to use the *metadata* to check if each chunk is big enough to hold the allocation. You can experiment with different ways to choose a matching chunk - possible choices include *first-fit*, which returns the first chunk found that is big enough, or *best-fit*, which returns the smallest suitable chunk. Make sure to unlink the returned chunk from the free list.
4. By this point, you should be able to pass all the tests! If you don't, you'll have to debug your implementation; printing out the allocations and using `gdb` to inspect the state of your heap can be very useful tools here.

Hand in only the file `mymalloc.c`.

Question 3 – Reference Counting [40%]

The program in the `q3` directory can be built using `make` while in that directory. You can then run the program by typing

```
./main A B C D
```

The program takes an arbitrary number of string arguments. It is *supposed* to print out a binary search tree, which is formed out of a subset of arguments, twice. However, as you can clearly see, the program does not produce this output; it correctly prints only one tree. **It has a bug.** To see the correct output of the program, you can remove all the `free()` calls in the program, replacing the dangling pointer bug with a memory leak.

The implementation uses two libraries, `list` and `tree`, implementing a doubly-linked list and a binary search tree respectively. Both libraries are free of bugs if used independently, but exhibit a bug when used together.

1. Find the bug and write a careful description of it in the file `q3.txt`. The bug is a dangling pointer somewhere. You may also want to read the header files carefully to see whether the libraries are being misused.
2. 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 can change the interface in `element.h` and any of the implementation in `element.c` (retaining its ability to store strings and numbers). You should not change any of the other `.h` files, and make minimal changes to the other `.c` files aside from adding reference counting (e.g. calls to element-specific `keep_ref/free_ref` functions that you define). Elements must still be shared between the list and tree - you should not simply make copies of the elements (which would introduce inefficiency).
3. After fixing the bug, test your program to ensure that it produces the correct output and that `valgrind` reports no memory leaks. 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 ./main <arguments>
```

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
```

What to Hand In

Use the `handin` program.

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

1. (optional) `PARTNER.txt` containing your partner's CWL login id and nothing else. 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` and `q3` just like you see in the code file;
4. in the `q2` directory: `mymalloc.c`;
5. in the `q3` directory: `q3.txt`, `element.c`, `element.h`, `list.c`, `main.c`, and `tree.c`.