15-122: Principles of Imperative Computation, Spring 2023

Written Homework 10

Due on Gradescope: Monday 3rd April, 2023 by 9pm

Name: _____

Andrew ID: _____

Section: _____

This written homework provides practice with some introductory C concepts.

Preparing your Submission You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- ***PDFescape*** or ***DocHub***, two web-based PDF editors that work from anywhere.
- ***Acrobat Pro***, installed on all non-CS cluster machines, works on many platforms.
- ***iAnnotate*** works on any iOS and Android mobile device.

There are many more — use whatever works best for you. If you'd rather not edit a PDF, you can always print this homework, write your answers *neatly* by hand, and scan it into a PDF file — *we don't recommend this option, though.*

Caution Recent versions of Preview on Mac are buggy: annotations get occasionally deleted for no reason. **Do not use Preview as a PDF editor.**

Submitting your Work Once you are done, submit this assignment on Gradescope. ***Always check it was correctly uploaded.*** You have unlimited submissions.

Question:	1	2	3	Total
Points:	2	4.5	6	12.5
Score:				

2pts 1. **Contracts in C**

The code below is taken from the lecture notes on hash sets in C0. This is also legal C code (assuming all the right definitions are available), but the contracts will not be checked in C.

```
elem hset_lookup(hset* H, elem x)
//@requires is_hset(H);
//@requires x != NULL;
//@ensures \result==NULL || elem_equiv(\result, x);
{
  int i = elemhash(H, x);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {
    //@assert p->data != NULL;
    if (elem_equiv(p->data, x)) {
      return p->data;
    }
  }
  return NULL;
}
```

In the spaces on the next page, translate the C0 contracts above into C contracts (as seen in class). You may need to make slight changes along the way. When compiled with the `-DDEBUG` flag, your translated code should behave exactly in the same way as the above C0 code when compiled with the flag `-d`. Use the variable `result` and do *not* simplify any C contracts even if it is immediately obvious from the context that you could do so.

© Carnegie Mellon University 2023

```
elem hset_lookup(hset *H, elem x) {




  elem result;
  int i = elemhash(H, x);
  for (chain* p = H->table[i]; p != NULL; p = p->next) {



    if (elem_equiv(p->data, x)) {







    }
  }




}
```

© Carnegie Mellon University 2023

2. **Free-range Geese**

You are a field biologist tracking geese migrations. You inherited some C code you hope will help you do this on a goose-by-goose fashion. However, this code is plagued with problems.

2pts **2.1** The `main` function below uses the following type declarations:

```
struct reading_header {
  int x;            // position in 3D
  int y;
  int z;
  int heading;      // direction: angle from North
};
typedef struct reading_header reading;

struct reading_node {
  reading *data;
  reading_list *next;
};
typedef struct reading_node reading_list;
```

The following code creates a test goose and populates some fields.

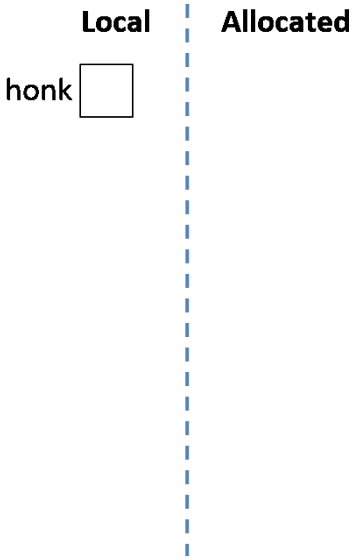
```
14 int main() {
15   reading_list *honk = xmalloc(sizeof(reading_list));
16   honk->data = xcalloc(sizeof(reading), 1);
17   honk->next = xmalloc(sizeof(reading_list));
18   honk->next->next = xmalloc(sizeof(reading_list));
19   honk->next->next->data = xmalloc(sizeof(reading));
20   honk->next->next->data->x = 13;
21   honk->next->next->data->y = 7;
22   honk->next->next->data->heading = 122;
23
24   // Memory Diagram 1
25   free(honk->next->next->next);
26   free(honk->next->next);
27   free(honk->next->data);
28   free(honk->data);
29   free(honk);
30
31   // Memory Diagram 2
32   return 0;
33 }
```

© Carnegie Mellon University 2023

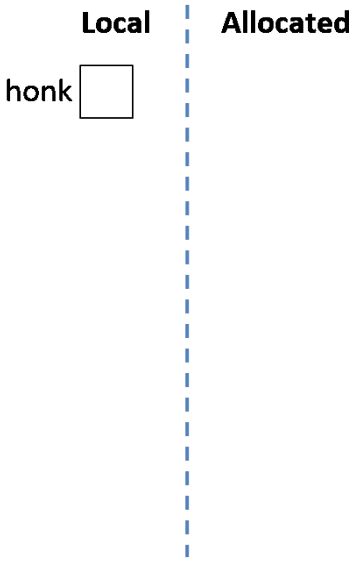
You suspect that even this simple test code has undefined behavior and memory leaks! Trace its execution, drawing the diagram of allocated memory on lines 24 and 31. Report any line that causes an undefined behavior, but then ignore it as you continue tracing execution. If a value is uninitialized, leave its box blank. In the second diagram, cross out any memory that was freed.

Cite the lines with undefined behavior: _____

Memory diagram 1 (as execution reaches line 24):



Memory diagram 2 (as execution reaches line 31):



(Did you remember to cross out the memory that was freed?)

© Carnegie Mellon University 2023

The code you inherited uses a makeshift data storage library implemented using generic linked lists. *(Note that these lists are distinct from the* `reading_list` *type used in the previous task.)* Here are the library-side type definitions.

```
typedef void* item;                    // Stored items (generic)
typedef void free_fn(item x);          // Function that frees an item

struct list_node {                     // Standard linked lists
  item data;                           // != NULL
  list *next;
};
typedef struct list_node list;

struct datadump_header {
  list *head;                          // NULL-terminated
  free_fn *free_elem;                  // NULL to leave items alone
};
typedef struct datadump_header datadump;
typedef datadump* datadump_t;

bool is_datadump(datadump *D);         // Checks that D is valid
```

You want to extend this library with the function `datadump_to_readings(D)` which moves all the items in the data dump `D` into a `reading_list`. This call disposes of the datadump `D`, which you shall assume was heap-allocated.

The next exercises are about memory leaks. Recall that memory is leaked when there are no more references to it. Memory that is returned to the user is not leaked. Consider the following example

```
1 int *f() {
2   int* p = xmalloc(sizeof(int));
3   p = xcalloc(1, sizeof(int));
4   return xmalloc(sizeof(int));
5 }
```

The memory allocated at line 2 becomes inaccessible at line 3 and is therefore leaked. The memory allocated at line 3 becomes inaccessible when we return from this function on line 4. The memory allocated at line 4 is not leaked as it is returned to the caller. We fix these leaks by inserting `free(p)` both between lines 2 and 3 and between lines 3 and 4.

© Carnegie Mellon University 2023

1pt

2.2 Your first attempt at writing `datadump_to_readings` is as follows. You suspect it leaks memory and want to fix the leaks.

```
27 reading_list *datadump_to_readings(datadump *D) {
28   REQUIRES(is_datadump(D));
29
30   reading_list *dummy = xcalloc(sizeof(reading_list), 1);
31   reading_list *prev = dummy;
32   list *l = D->head;
33   free(D);
34
35   while (l != NULL) {
36     list *nxt = l->next;
37     prev->next = xcalloc(sizeof(reading_list), 1);
38     prev->next->data = (reading*)l->data;
39     prev = prev->next;
40     l = nxt;
41   }
42   reading_list *res = dummy->next;
43   return res;
44 }
```

In the table below, record the leaks in this code. For each leak, write down the line which causes the memory to become inaccessible. Then, write a line of code that fixes it, and indicate where to insert it. (*You may not need all spaces.*)

Line	**Code that fixes it**	**Where to insert the fix**
		between lines _____ and _____
		between lines _____ and _____
		between lines _____ and _____
		between lines _____ and _____

© Carnegie Mellon University 2023

With `datadump_to_readings` fixed, here's the resulting extended interface to the data storage library, with C0-style contracts for readability.

```
// typedef __________ *datadump_t;
typedef void* item;                 // Stored items (generic)
typedef void free_fn(item x);       // Function that frees an item

datadump_t datadump_new(free_fn *F)
/*@ensures \result != NULL; @*/ ;

void datadump_insert(datadump_t D, item x)
/*@requires D != NULL && x != NULL; @*/ ;

reading_list *datadump_to_readings(datadump *D)
/*@requires D != NULL; @*/ ;
```

As a client, you have already written the functions

```
bool more_readings(int id) ;

reading *get_next_reading(int id)
/*@requires more_readings(id); @*/ ;

void print_readings(reading_list *R) ;
```

The first returns whether there are more location readings to process for goose number `id`. If so, the second stores the location values of the next unprocessed reading on the heap and returns a pointer to it (behind the scenes, it marks this reading as processed). The last one prints the contents of a `reading_list*`.

You write the following test that combines all these functionalities.

```
83 #define HONK 15122
84 int main() {
85   datadump_t D = datadump_new(NULL);
86   while (more_readings(HONK))
87     datadump_insert(D, (void*)get_next_reading(HONK));
88   }
89
90   reading_list *R = datadump_to_readings(D);
91   print_readings(R);
92   free(R);
93   return 0;
94 }
```

© Carnegie Mellon University 2023

0.5pts **2.3** The first time you run it, `valgrind` tells you you are still leaking memory! In one line, what is the problem?

1pt **2.4** Write a **recursive** function `free_readings(R)` to be used in place of line 92 so that no memory is leaked. You may assume that all the memory accessible from `R` is heap-allocated, and no data field is `NULL`. You will need to refer to the type declarations on page 3.

```
void free_readings(reading_list* R) {




}
```

© Carnegie Mellon University 2023

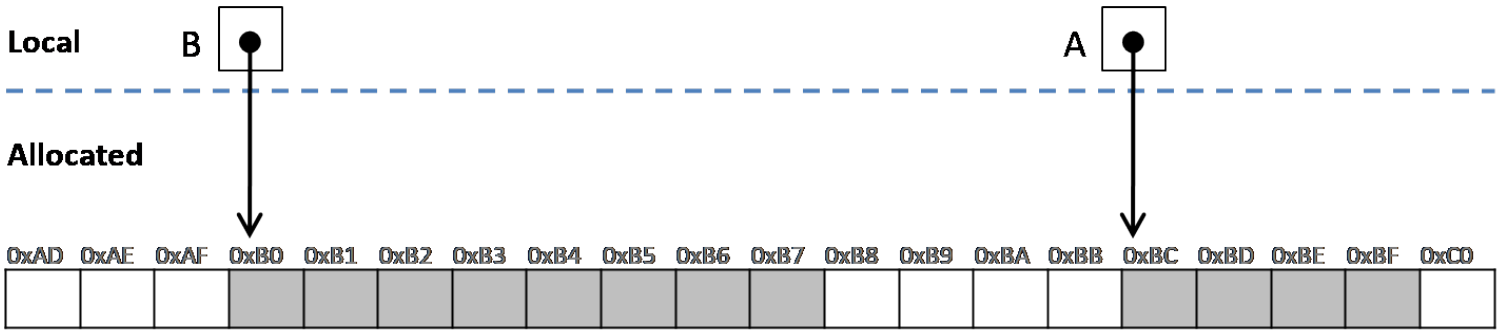
3. **A Heap of Bytes**

We can think of the heap as a huge block of memory administered by the system. When we call `malloc` and `calloc` (or their `x` variants), we get a temporary "permit" to use some of it. Specifically, `malloc` reserves for us a chunk of this memory and returns a pointer to this block, which is now ours. When we are done with it, we call `free` to give it back — our permit is not valid any more.

2pts **3.1** Consider the following simple C program:

```
1 int main() {
2   int *A = xmalloc(sizeof(int));
3   int *B = xcalloc(2, sizeof(int));
4   int *C = B + 1;
5   *A = 5;
6   *C = 6;
7   return 0;
8 }
```

Here's an abstract representation of the heap just after line 3 has been executed (assuming an **`int`** takes up 4 bytes). The shaded regions were reserved for you. The other regions belong to the system and you have no right to use them.



Based on this representation, what is the value of the following expressions just before this code executes line 7 (the **`return`** statement)? If evaluating an expression would have undefined behavior, put down `0x15122`. For this task, you may assume execution will not crash if you access any of these addresses.

`A:`		`*A:`	
`B:`		`*B:`	
`C:`		`*(B+1):`	
`B+1:`		`*(B+2):`	
`B+2:`		`*((int*)0xB4):`	[1]

[1] Do not do this in your code!

© Carnegie Mellon University 2023

1pt **3.2** Let's say we free `A` just before line 7. Answer the following questions.

- List the addresses in this heap fragment that would be undefined to access. (Feel free to write address ranges.)

  _____

- How many bytes of memory will valgrind report as lost once this program returns?

  _____

- Say we add the line

  ```
  int v = *A;
  ```

  right after freeing `A`. Check all the behaviors that may plausibly take place as a result of executing this line.

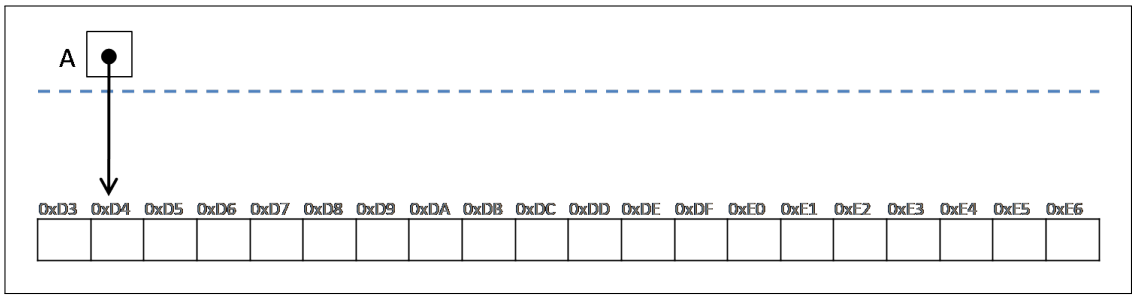
  - ☐ `v` contains `0`.
  - ☐ `v` contains `0x00000005`.
  - ☐ sinks Christopher Columbus's ships before he landed in the Americas in 1492.
  - ☐ turns off the sun within a second of executing this instruction.
  - ☐ the program halts with a segmentation fault.

© Carnegie Mellon University 2023

1pt **3.3** Calling `malloc` or `calloc` (or their `x` variants) procures a block of contiguous memory of the requested length (in bytes). Because it is contiguous, we can treat it as an array. Consider the following code:

```
21 void f(int *A, int *B, size_t n) {
22   for (size_t i = 0; i < n; i++)
23     *(B + i) = *(A + i);
24 }
25
26 int main() {
27   size_t n = 4;
28   int *A = xcalloc(n, sizeof(int));
29   for (size_t i = 0; i < n; i++) {
30     *(A + i) = i;
31   }
32
33   f(A, A + 2, n/2);
34   free(A);
35   return 0;
36 }
```

In the picture below, highlight the bytes in memory that get allocated on line 28. Then, clearly indicate on the diagram which addresses correspond to each of the pointers `A+1`, `A+2`, `A+3`. Again, assume an **`int`** takes up 4 bytes. Feel free to write numbers in the boxes as you trace the code.



1pt **3.4** What can we say after the function call on line 33 returns?

`A` contains [_____, _____, _____, _____]

How many bytes get freed on line 34? _____

© Carnegie Mellon University 2023

1pt **3.5** While C lets you use pointer arithmetic in your code, programs are more readable if we only ever use the standard array notation. Rewrite the previous function so that it does not use pointer arithmetic, leaving everything else the same.

```
void f(int *A, int *B, size_t n) {
  for (size_t i = 0; i < n; i++)

    _____________________________________________________________;
}

int main() {
  size_t n = 4;
  int *A = xcalloc(n, sizeof(int));
  for (size_t i = 0; i < n; i++) {

    _____________________________________________________________;
  }

  _____________________________________________________________;
  free(A);
  return 0;
}
```

© Carnegie Mellon University 2023