15-122: Principles of Imperative Computation, Spring 2020

Written Homework 11

Due on Gradescope: Thursday 16th April, 2020 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.
- ***Preview***, the Mac's PDF viewer.
- ***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.*

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:	3.5	4.5	4	12
Score:				

3.5pts

1. **Contracts in C**

   The code below is taken from the lecture notes on binary search trees 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 tree_lookup(tree* T, elem x)
//@requires is_tree(T) && x != NULL;
//@ensures \result == NULL || elem_compare(\result, x) == 0;
{
  if (T == NULL) return NULL;
  int cmp = elem_compare(x, T->data);
  if (cmp == 0) {
    return T->data;
  } else if (cmp < 0) {
    return tree_lookup(T->left, x);
  } else {
    //@assert cmp > 0;
    return tree_lookup(T->right, x);
  }
}

elem set_lookup(set* B, elem x)
//@requires is_set(B) && x != NULL;
//@ensures \result == NULL || elem_compare(\result, x) == 0;
{
  return tree_lookup(B->root, x);
}
```

   Rewrite the function in the box on the next page as follows:

   - Insert assignment statements so that all return statements have the form **`return`** `result`. (In other words, use the variable `result`, defined on the next page, to hold the return value for all cases and use this variable in your postcondition.)
   - Insert any necessary C contracts so that, when compiled with the flag `-DDEBUG`, contracts will be checked as they would be in C0 with the flag `-d`.

   Do *not* simplify any contracts even if it is immediately obvious from the context that you could do so. You may omit the C0 contracts (lines beginning `//@`) even though in practice we might like to keep them.

```
elem tree_lookup(tree* T, elem x)


  elem result;


}

elem set_lookup(set* B, elem x)


  elem result;


}
```

4.5pts

2. Allocating and Freeing Memory in C

Here is a leaky C program that works with NULL-terminated linked lists. We've omitted the code for `print_list` because it can't leak any memory. Contracts have been omitted for the sake of space.

```
typedef struct list_node list;
struct list_node {
  int data;
  list* next;
};

void free_list(list* L)
{
  list* current = L;

  while (current != NULL)
  {
    list* next = current->next;
    free(current);
    current = next;
  }
  return;
}

void sum(list* L)
{
  list* sum = xmalloc(sizeof(list));
  sum->data = 0;
  sum->next = NULL;
  list* current = L;

  while (current != NULL)
  {
    sum->data += current->data;
    current = current->next;
  }

  L->data = sum->data;
  L->next = NULL;

  return;
}
```

```
69 int main()
70   {
71   list* current = NULL;
72   for (int i=0 ; i<10 ; i++)
73   {
74     ASSERT(0 <= i);
75     list* new = xmalloc(sizeof(list));
76     new->data = i;
77     new->next = current;
78     current = new;
79   }
80   printf("Initial list: ");
81   print_list(current);
82   sum(current);
83   printf("Summed list: ");
84   print_list(current);
85 
86   return 0;
87 }
```

In the table below, give the line number of each line that leaks memory *(you may not need all rows)*. A line is considered to leak memory if, as a result of executing it, some allocated memory has not been freed, and no further references to that memory are possible. Returning from the `main` function without deallocating everything that was allocated is considered a leak (even though the operating system will clean it up).

Indicate how to fix the leak(s) by writing any extra code that needs to be added, with the line numbers between which it should be inserted. Your changes should not alter the behavior of the program other than fixing the leaks.

Line number of leak	Code that fixes it	Where to insert it
______	______________________________	______
______	______________________________	______
______	______________________________	______
______	______________________________	______
______	______________________________	______

4pts 3. **Pass by Reference and Arrays versus Pointers in C**

The following little program allocates and initializes an array of numbers, then calls a function f on two of its elements. Rewrite these functions in the box below so that it has the same behavior, uses the same variables at any point, but doesn't use the pointer arithmetic notation. You may not insert additional instructions.

```
#include <stdlib.h>
#include <stdio.h>
#include "lib/xalloc.h"
#include "lib/contracts.h"

void f(char *x, char *y);   // Code omitted

char *mk_char_array(size_t n) {
  return xmalloc(sizeof(char) * n);
}

int main() {
  char *A = mk_char_array(110);
  for (int i = 0 ; i < 10 ; i++) {
    ASSERT(0 <= i);
    *(A + i) = 42 - i;
  }
  char *TMP = A+3;
  ASSERT(*(A+2) == 40);
  ASSERT(*(TMP+7) == 32);
  f(A+2, A+7);
  ASSERT(*(TMP+-1) == 32);
  ASSERT(*(A+4) == 40);

  printf("All tests passed.\n");
  free(A);
  return 0;
}
```

```
char *mk_char_array(size_t n) {



}

int main() {
















}
```