15-122: Principles of Imperative Computation, Fall 2017

Written Homework 11

Due: Thursday 16th November, 2017 by 9pm

Name: Shaojie Zhang

Andrew ID: shaojiez

Section: O

This written homework provides practice with some introductory C concepts.

Instructions

You can prepare your submission in one of two ways:

Just edit (preferred) Use any PDF editor (e.g., Preview on Mac, iAnnotate on mobile, Acrobat Pro installed on all non-CS cluster machines and most platforms) to typeset your answers in the given spaces — you can even draw pictures. *That's it.*

Print and Scan Alternatively, print this file, write your answers *neatly* by hand, and then scan it into a PDF file. *This is pretty labor-intensive.*

Once you have prepared your submission, submit it on Gradescope. You have unlimited submissions.

Question:	1	2	3	Total
Points:	3.5	4.5	4	12
Score:				

Evaluation Summary Once this homework is graded, you will be able to find a summary of your performance on Gradescope.

3.5pts 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;
}
```

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 hset_lookup(hset* H, elem x) {
    REQUIRES(is_hset(H));
    REQUIRES(x != NULL);
    int i = elemhash(H, x);









  elem result;
   for (chain* p = H->table[i]; p != NULL; p = p->next) {
        ASSERT (p->data != NULL);
        if (elem_equiv(p->data, x)) {
            result = p->data;
            ENSURES (result == NULL || elem_equiv(result, x));
            return result;
        }
   result = NULL;
   ENSURES(result == NULL || elem_equiv(result, x));
   return 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.

```
1  typedef struct list_node list;
2  struct list_node {
3    int data;
4    list* next;
5  };
6  void free_list(list* L) {
7    list* current = L;
8    while (current != NULL) {
9      list* next = current->next;
10     free(current);
11     current = next;
12   }
13   return;
14 }
15 void sum(list* L) {
16   list* sum = xmalloc(sizeof(list));
17   sum->data = 0;
18   list* current = L;
19   while (current != NULL) {
20     sum->data += current->data;
21     current = current->next;
22   }
23   L->data = sum->data;
24   L->next = NULL;
25   return;
26 }
27 int main() {
28   list* current = NULL;
29   for (int i=0 ; i<10 ; i++) {
30     ASSERT(0 <= i);
31     list* new = xmalloc(sizeof(list));
32     new->data = i;
33     new->next = current;
34     current = new;
35   }
36   printf("Initial list: "); print_list(current);
37   sum(current);
38   printf("Summed list: "); print_list(current);
39   return 0;
40 }
```

In the table below, give the line number of each line that leaks memory. 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.

Line number of leak	Code that fixes it	Where to insert it
25	free(sum);	24, 25
39	free_list(current);	38, 39
24	free_list(L->next);	23, 24

3. **Pass by Reference Using C**

At various points in our C0 programming experience, we had to use somewhat awkward workarounds to deal with functions that need to return more than one value. Stack-allocated data structures and the address-of operator (`&`) in C give us a new way of dealing with this issue.

Sometimes, a function needs to be able to both 1) signal whether it can return a result, and 2) return that result if it is able to. Consider the following function `parse_string` that attempts to parse a string into an integer:

```
bool parse(char *s, int *i);  // Returns true iff parse succeeds

void parse_string(char *s) {
  REQUIRES(s != NULL);
  int *i = xmalloc(sizeof(int));
  if (parse(s, i))
    printf("Success: %d.\n", *i);
  else
    printf("Failure.\n");
  free(i);
  return;
}
```

The function `parse_string` relies on `parse`, a function which both sets `*i` to an integer equivalent to the integer pattern in `*s` (if possible) and also returns a boolean value of `true` if the parse succeeds, or `false` otherwise.

2pts **3.1** Using the address-of operator, rewrite the body of the `parse_string` function so that it does not heap-allocate, free, or leak any memory on the heap. You may assume `parse` has been implemented (its prototype is given above).

```
void parse_string(char *s) {
  REQUIRES(s != NULL);
   int i = 0;
   if (parse(s, &i))
       printf("Success: %d.\n", *i);
   else
       printf("Failure.\n");




  return;
}
```

2pts

3.2 In both C and C0, multiple values can be 'returned' by bundling them in a struct:

```
struct bundle {
  int fst;
  int snd;
};

struct bundle *split_int(int p) {
  struct bundle *A = xmalloc(sizeof(struct bundle));
  A->fst = p>=0 ? 1 : -1;  // first value to be returned
  A->snd = abs(p);         // second value to be returned
  return A;            // return both values together as a struct
}

int main() {
  ...
  struct bundle *B = split_int(-42);
  int sign  = B->fst;
  int value = B->snd;
  free(B);
  ...
}
```

Complete the declaration of the function `split_int`, as well as the snippet of `main`, to avoid heap-allocating, freeing, or leaking any memory on the heap. The rest of the code (`...`) should continue to behave exactly as it did before.

```
void split_int(___struct bundle *A___, int p) {
  A->fst = p>=0 ? 1 : -1;
  A->snd = abs(p);
  return;
}

int main() {
  ...
  struct bundle B;

  split_int(_____&B_____, -42);

  int sign  = _____B.fst_____;

  int value = _____B.snd_____;
  ...
}
```