15-122 : Principles of Imperative Computation, Fall 2015

Written Homework 6

Due: Monday, October 12, 2015

Name: ____________________

Andrew ID: ____________________

Section: ____________________

This written homework covers structs, pointers, and linked lists.

Print out this PDF double-sided, staple pages in order, and write your answers on these pages *neatly*.

The assignment is due at the beginning of recitation on Monday, October 12, 2015.

You must hand in your homework yourself; do not give it to someone else to hand in.

(This page intentionally left blank. Remember to print double-sided!)

Question	Points	Score
1	4	
2	2	
3	4	
Total:	10	

1. **Multiple Return Values**

One of the problems that both structs and pointers can solve in different ways is the problem of returning more than one piece of information from a function. For instance, a function that tries to parse a string as an integer needs to return both the successfully-parsed integer and information about whether that parse succeeded. Because *any* number could be the result of a successful parse, if the function only returns an `int`, there's no way to distinguish a failed parse from a successful one.

```
int parse_int(string str) {
   int n;
   bool parse_successful;
   ...
   if (parse_successful) return n;
   return ???; /* What do we do now? */
}
```

In each of the following exercises, a main function wants to print the value that `parse_int` is storing in the assignable variable `n`, but *only* when the boolean value stored in `parse_successful` is `true`; otherwise we want to print out *"Parse error"*.

You don't have to use all the blank lines we have provided, but you shouldn't use any extra lines. **Double-check your syntax; we will be picky about syntax errors for this question.**

2pts (a) Finish this program so that the code will parse the first command-line argument as an int if possible. Make sure all your pointer dereferences are provably safe.

```
int* parse_int(string str) {
    int n;
    bool parse_successful;
    // Omitted code that tries to parse the string. Puts the
    // result in the local variable n and sets parse_successful
    // to true if it can, otherwise sets parse_successful to false.

    if (parse_successful) {

        _______________________________________________;

        _______________________________________________;

        return ____________________________________;
    }
    return ________________________________________;
}
```

```
int main() {
    args_t A = args_parse();
    if (A->argc != 1) error("Wrong number of arguments");
    int* p = parse_int(A->argv[0]);

    if (__________________) printint(__________________);
    else error("Parse error");
    return 0;
}
```

2pts (b) Complete another program that works the same way, but that gives a different type to `parse_int`. The missing argument should be a pointer. Make sure all your pointer dereferences are provably safe.

```
bool parse_int(string str, ______________________________)

//@requires ____________________________________________;
{
    int n;
    bool parse_successful;
    // Same omitted code...

    if (parse_successful) {

        __________________________________________________;

        return ______________________________________;
    }
    return ____________________________________________;
}

int main() {
    args_t A = args_parse();
    if (A->argc != 1) error("Wrong number of arguments");

    __________________________________________________;

    bool res = parse_int(A->argv[0], ___________________);

    if (__________________) printint(__________________);
    else error("Parse error");
    return 0;
}
```

2. **Reasoning with Linked Lists**

You are given the following C0 type definitions for a linked list of integers.

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

struct list_header {
    list* start;
    list* end;
};
typedef struct list_header* linkedlist;
```

An empty list consists of one `list_node`. All lists have one additional node at the end that does not contain any relevant data, as discussed in class.

In this task, we ask you to analyze a list function and reason that each pointer access is safe. You will do this by indicating the line(s) in the code that you can use to conclude that the access is safe. Your analysis must be precise and minimal: only list the line(s) upon which the safety of a pointer dereference depends. If a line does not include a pointer dereference, indicate this by writing NONE after the line in the space provided. As an example, we show the analysis for an `is_segment` function below.

```
          bool is_segment(list* s, list* e) {
/*  1 */      if (s == NULL) return false;          NONE
/*  2 */      if (e == NULL) return false;          NONE
/*  3 */      if (s->next == e) return true;        1
/*  4 */      list* c = s;                          NONE
/*  5 */      while (c != e && c != NULL) {         NONE
/*  6 */          c = c->next;                      5
/*  7 */      }                                     NONE
/*  8 */      if (c == NULL)                        NONE
/*  9 */          return false;                     NONE
/* 10 */      return true;                          NONE
          }
```

When we reason that a pointer dereference is safe, *only* that dereference is okay. So, in the example below, we have to use line `81` to prove both line `82` and line `83` safe.

```
/* 81 */  //@assert is_segment(a, b);
/* 82 */  a->next = b;
/* 83 */  list* l = a->next;
```

We don't allow you to say that, because line 82 didn't raise an error, `a` must not be `NULL` and therefore `83` must be safe. (This kind of reasoning is error-prone in practice.)

Here's a mystery function:

```
/* 11 */  void mystery(linkedlist a, linkedlist b)
/* 12 */  //@requires a != NULL;                        ___NONE______
/* 13 */  //@requires b != NULL;                        ___NONE______
/* 14 */  //@requires is_segment(a->start, a->end);     ______________
/* 15 */  //@requires is_segment(b->start, b->end);     ______________
/* 16 */  {
/* 17 */    list* t1 = a->start;                        ______________
/* 18 */    list* t2 = b->start;                        ______________
/* 19 */    while (t1 != a->end && t2 != b->end)        ______________
/* 20 */    //@loop_invariant is_segment(t1, a->end);   ______________
/* 21 */    //@loop_invariant is_segment(t2, b->end);   ______________
/* 22 */    {
/* 23 */        list* t = t2;                           ______________
/* 24 */        t2 = t2->next;                          ______________
/* 25 */        t->next = t1->next;                     ______________
/* 26 */        t1->next = t;                           ______________
/* 27 */        t1 = t1->next->next;                    ______________
/* 28 */     }
/* 29 */     b->start = t2;                             ______________
/* 30 */  }
```

1pt (a) Explain why line 19 is safe: first, clearly state what the conditions for the safety of line 19 are, and second, explain why we know those lines are safe.

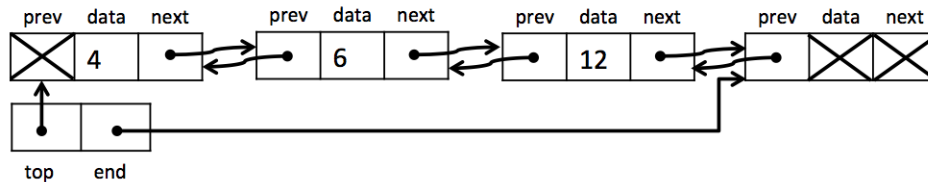
1pt (b) Why can we not use the combination of line 14 (which tells us that `a->start` is not `NULL`) and line 17 (which tells us that `t1` is `a->start`) to reason that `t1` is not `NULL` and therefore that line 26 is safe? Why do we actually know line 26 is safe?

3. **Doubly-Linked Lists**

Consider the following interface for `stack` that stores elements of the type `elem`:

```
typedef struct stack_header* stack_t;

bool stack_empty(stack_t S)       /* O(1) */
  /*@requires S != NULL; @*/;

stack_t stack_new()               /* O(1) */
  /*@ensures \result != NULL; @*/
  /*@ensures stack_empty(\result); @*/;

void push(stack_t S, elem x)    /* O(1) */
  /*@requires S != NULL; @*/;

elem pop(stack_t S)             /* O(1) */
  /*@requires S != NULL; @*/
  /*@requires !stack_empty(S); @*/;
```

Suppose we decide to implement the stack (of integers) using a doubly-linked list so that each list node contains two pointers, one to the next node in the list and one to the previous (prev) node in the list:




```
typedef struct list_node list;
struct list_node {
    elem data;
    list* prev;
    list* next;
};

typedef struct stack_header stack;
struct stack_header {
    list* top;
    list* bottom;   // points to dummy node
};
```

The top element of the stack will be stored in the first (head) node of the list, and the bottom element of the stack will be stored in the second-to-last node in the list, with the last node being a "dummy node."

An empty stack consists of a dummy node only: the `prev`, `data`, and `next` fields of that dummy are all unspecified. A non-empty stack has an unspecified `prev` field for the top, and an unspecified `data` and `next` field for the bottom.

2pts

(a) Modify the singly-linked list implementation of stacks given below to work with the doubly-linked list representation given above. For each function, either state the modification(s) that need to be made (e.g. "Insert the statement XXXX after line Y," "Remove line Z," "Change line Z to XXXX," etc.) or state "No change needs to be made." You may assume there is an appropriate `is_stack` specification function already defined. Be sure that your modifications still maintain the O(1) requirement for the stack operations.

```
/*  1 */  stack* stack_new()
/*  2 */  //@ensures is_stack(\result);
/*  3 */  //@ensures stack_empty(\result);
/*  4 */  {
/*  5 */      stack* S = alloc(struct stack_header);
/*  6 */      list* L = alloc(struct list_node);
/*  7 */      S->top = L;
/*  8 */      S->bottom = L;
/*  9 */      return S;
/* 10 */  }
```

Solution: *No change needs to be made.*

(Explanation: according to the description above, the empty stack contains one dummy node with all fields unspecified.)

```
/* 11 */  bool stack_empty(stack* S)
/* 12 */  //@requires is_stack(S);
/* 13 */  {
/* 14 */      return S->top == S->bottom;
/* 15 */  }
```

Solution:

```
/* 16 */  void push(stack* S, elem x)
/* 17 */  //@requires is_stack(S);
/* 18 */  //@ensures is_stack(S);
/* 19 */  {
/* 20 */      list* L = alloc(struct list_node);
/* 21 */      L->data = x;
/* 22 */      L->next = S->top;
/* 23 */      S->top = L;
/* 24 */  }
```

Solution:

```
/* 25 */  elem pop(stack* S)
/* 26 */  //@requires is_stack(S);
/* 27 */  //@requires !stack_empty(S);
/* 28 */  //@ensures is_stack(S);
/* 29 */  {
/* 30 */      elem e = S->top->data;
/* 31 */      S->top = S->top->next;
/* 32 */      return e;
/* 33 */  }
```

Solution:

1pt

(b) We wish to add a new operation `stack_bottom` to our stack implementation from the previous part.

```
elem stack_bottom(stack_t S)    /* O(1) */
  /*@requires S != NULL && !stack_empty(S); @*/ ;
```

This operation returns (but does not remove) the bottom element of the stack. Write an implementation for this function using the doubly-linked list implementation of stacks from the previous part. Be sure that your function runs in constant time. *(Remember that the linked list that represents the stack has a dummy node.)*

Solution:

```
elem stack_bottom(stack* S)
//@requires is_stack(S);
//@requires !stack_empty(S);
{

}
```

1pt

(c) Now, consider the following broken implementation of `is_stack` for this stack implementation.

```
bool is_segment(list* node1, list* node2) {
  if (node1 == NULL) return false;
  if (node1 == node2) return true;
  return is_segment(node1->next, node2);
}

bool is_stack(stack* S) {
  return S != NULL && is_segment(S->top, S->bottom);
}
```

Draw a complete picture of a stack data structure (with integer elements) that contains at least 4 allocated `list_node` structs and that returns `true` from `is_stack` yet would not be well-formed. *Give specific values everywhere.* ***Don't*** *use Xs anywhere; they are for unspecified values. So your diagram should depict pointers (possibly NULL) and integers.* For full credit your example struct must fail the unit test below with a segfault or an assertion failure after passing the initial assertion.

Stack picture:

```
// Unit test that your example above should fail
int main() {
    stack* S = // code that constructs the example above
               // by necessity, this won't respect the interface
    assert(is_stack(S) && !stack_empty(S)); // This must pass
    elem x = stack_bottom(S);
    elem y = pop(S);
    while (!stack_empty(S)) {
        y = pop(S);
        assert(is_stack(S));
    }
    assert(x == y);
    return 0;
}
```