Homework 1 is due Thursday

Theory: Due in class.
Coding: Due at 11:59 PM, unless you’re taking a late day. (I strongly recommend against using late days early. You will need them later.)

Take advantage of office hours and cluster hours!

How to Debug C₀

Here are some bugs you’ll likely encounter while coding in C₀, as well as some strategies to make debugging less painful.

Static Errors
These are the errors that happen as you try to compile your programs. A static error means that your code is ill-formed. They’re my favorite, since they’re the easiest to fix. They will be one of the following:

- Lexical errors, like 09e3f3, which is not a legal variable, function name, or value
- Syntax errors, like min(x y), which is an ill-formed function call
  - Unexpected end of file or eof - most likely, you’ve got a mismatched {, (, [
  - Indentation - if you’re using Emacs, use the tab key to indent your files. You might find some surprises that result in an infinite loop or two.
- Type errors, like (39 + "happy birthday"), which is an attempt to add an int to a non-int value
- Static semantic errors, like using a variable before it is initialized

Dynamic Errors
These problems don’t show up until your code is running. They’re trickier to catch because C₀ doesn’t always complain loudly when they arise.

- Runtime errors, like 1/0. C₀ will raise an exception for these.
  - Arithmetic exceptions - div by 0, dividing min_int by -1
    * If you have x / y, check for invalid values with //@assert y != 0
    * If you have A[i], check for bad indexing with //@assert 0 <= i && i <= \length(A)
    * If you’re dereferencing a pointer p, check that it’s not NULL with //@assert p != NULL
  - Memory exceptions - out-of-bounds array access, dereferencing null pointer, bad memory allocation
  - Contract Exceptions - only when run with -d flag. More helpful error messages
  - Library Exceptions - if an exception from a C library is raised. Check library documentation
- Logical errors, like calling foo(24) if foo requires that its argument is odd.
- Logical errors, where your program runs without failing but returns an incorrect result

Print statements can be helpful, if you want to see the value of your variables. Remember that prints are buffered, so nothing will show until you print a newline character.
Linear Search: Finding a value x in an array A with length n

```java
1 bool is_sorted(int[] A, int n)
2    { //@requires 0 <= n & n <= \length(A);
3        for (int i = 0; i < n-1; i++)
4            @loop_invariant (n == 0) || (0 <= i & i <= n-1);
5            if (!(A[i] <= A[i+1])) return false;
6        }
7    return true;
8 }
9 return true;
```

Some things to note for is_sorted:

- The precondition allows you to verify that the first n elements of the array are sorted, rather than all of it.
- n could be 0, so we account for that in the loop invariant, since LIs are true whether or not you’re in the loop.
- If the array is not sorted, the function terminates as soon as it encounters an out-of-order pair.

```java
1 int linsearch(int x, int[] A, int n)
2    { //@requires 0 <= n & n <= \length(A);
3        @requires is_sorted(A,n);
4        { for (int i = 0; i < n && A[i] <= x; i++)
5            @loop_invariant 0 <= i & i <= n;
6            @loop_invariant i == 0 || A[i-1] < x;
7            if (A[i] == x) return i;
8        }
9    return -1;
10 }
```

linsearch returns the index of the first occurrence of x in the first n elements of A, or -1 if not found.

Note: line 7 is an example of short-circuit evaluation. If the first condition is true, we don’t check the second. This shows that A || B and B || A aren’t always the same.

Look at line 7. If the LI holds at the start of a loop iteration, is it true at the end? Line 5 shows that it holds upon first entering the loop. After each iteration is complete, i’ = i + 1, so we show A[i’-1] < x, or rather, A[i] < x.
The loop iteration starts with A[i] <= x. If we get all the way through the iteration without returning, we know then that A[i] != x. Thus we can conclude A[i] < x.

Now we want some good postconditions for the function based on what the preconditions and loop invariants tell us. We know that the result that gets returned is either -1 or an integer in the interval [0, n-1], so we have

```java
//@ensures -1 <= \result && \result < n;
```

We also know that if the result is -1, then x isn’t one of the first n elements of A.

If our result is in [0,n-1], then we know that’s the first occurrence of x, so x can’t be in [0, \result - 1]. Thus we have

```java
//@ensures (
result == -1 && !is_in(x,A,n))
|| (A[\result] == x && !is_in(x,A,\result)); @*/
```