15-122: Principles of Imperative Computation

Recitation 4 — Josh Zimmerman

Debugging tips and tricks

`http://c0.typesafety.net/tutorial/Debugging-C0-Programs.html` has some very useful tips on debugging, and addressess common pitfalls. I'll summarize some of the points here. I highly encourage you to read over that site on your own time for more details.

- Compilation errors
  - *Lexical errors:* Invalid numbers or variable names, like `0a4`, generate an error.
  - *Syntax errors* are generated by sequences of characters that don't make sense, like `f(, 4);` or `x + 3 = 4;`
  - *Type errors* arise when you write expressions that don't make sense based on the types of variables, like `true == 3` or `3 + "hello"`.
  - *Unexpected EOF* errors are generally caused by an unmatched brace, parenthesis, or similar character. Most editors can be configured to highlight unmatched parens and braces.
  - *Undeclared variable errors* happen if you use a variable before declaring it.
- Runtime errors
  - *Floating point* or *division by zero* errors generally indicate that you divided by zero, or divided `int_min()` (`0x80000000`) by `-1`. They will also occur if you try to shift left or right by 32 or more or by a number less than 0.
  - *Segmentation faults* occur if you attempt to access memory that you can't access. Right now, the only thing we've covered that can cause this is out-of-bounds array access (accessing a negative index of an array or accessing something past the end of the array), but later we'll see that `NULL` pointer dereferences can also cause this.
  - *Contract errors* occur when a contract is violated and contract checking is turned on.
- Weird behavior with conditionals and loops: If some code that should be running in a conditional or loop isn't, make sure you have braces around the block. It's much harder to debug otherwise.

  ```
  while (some_condition)
      printint(i);
      print("\n");
  ```

  will only print the newline after `some_condition` is false. You should add braces before and after the loop to get correct behavior.
- Printing: C0 does not print anything until it sees a newline. This can cause things to get printed at unexpected times when you are debugging your program. You should ALWAYS print a newline after any string you print, using either `print("\n")` or `println("")`. `println` will work for any string: `println("Hello!")` is the same as `print("Hello!\n");`

Using contracts to debug is invaluable. If you can catch array out of bound errors or arithmetic before they happen, the extra information contract failures give you could save hours of debugging.

Print statements are also very useful to help investigate *why* your contracts are failing or your code is returning strange results. They let you examine the values of variables and see where things go wrong.

Another useful tactic is to use a small example, and see what your code does with it by evaluating your code *by hand*. When you evaluate by hand, you can see exactly where a mistake happens as soon as possible, allowing you to catch and fix it quickly.

Basic linear search: recap

(Note: I assume the `is_in` and `is_sorted` functions exist as defined in class.)

```
1  int lin_search(int x, int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@requires is_sorted(A, 0, n);
4  /*@ensures (-1 == \result && !is_in(x, A, 0, n))
5        || ((0 <= \result && \result < n) && A[\result] == x);
6   @*/
7  {
8     for (int i = 0; i < n; i++)
9     //@loop_invariant 0 <= i && i <= n;
10    //@loop_invariant !is_in(x, A, 0, i);
11    {
12       if (A[i] == x) {
13          return i; // We found what we were looking for!
14       }
15       else if (x < A[i]) {
16          return -1; // We've passed the last point it could be, so it's not there
17       }
18       //@assert A[i] < x;
19    }
20    return -1;
21 }
```

Now, let's look at this code and see if we can prove that it works. Work on your own or with other people to follow the four-step process to proving that linear search works. (Remember: Show that the loop invariants hold initially, that they are preserved, that the loop invariants and the negation of the loop condition imply the postcondition, and that the loop terminates.)

Binary search

Binary search lets us search arrays *substantially* more quickly than linear search does.

The basic idea behind binary search is that if we're searching for `x`, we look in the middle of a sorted array and compare that element to `x`. If that element is smaller than `x`, we look in the top half of the array and if that element is bigger than `x`, we look in the bottom half of the array. (If that element is equal to `x`, then we're done.)

We're going to work through a few examples on the board to illustrate the number of steps binary search takes on arrays in practice, but the theoretical view is as follows: On every iteration of the loop, we roughly cut in half the amount of the array that we still have to look at—at every step, we throw out half what's left of the array.

So, we look at half of the array, and we then look at half of that, and so on. How many halvings will it take until we're looking at 1 element?

Here's the code for binary search. We're going to look at a proof of its correctness.

```
1 int binsearch(int x, int[] A, int n)
2 //@requires 0 <= n && n <= \length(A);
3 //@requires is_sorted(A, 0, n);
4 /*@ensures (-1 == \result && !is_in(x, A, 0, n))
5       || ((0 <= \result && \result < n) && A[\result] == x);
6  @*/
7 {
8    int lower = 0;
9    int upper = n;
10   while (lower < upper)
11   //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
12   //@loop_invariant lower == 0 || A[lower-1] < x;
13   //@loop_invariant upper == n || A[upper] > x;
14   {
15      int mid = lower + (upper-lower)/2;
16      if (A[mid] < x) {
17         // We can ignore the bottom half of the array now, since we
18         // know that every thing in that half must be less than x
19         lower = mid+1;
20      } else if (A[mid] > x) {
21         // We can ignore the upper half of the array, since we know
22         // that everything in that half must be greater than x
23         upper = mid;
24      } else {
25         //@assert A[mid] == x;
26         return mid;
27      }
28   }
29   //@assert lower == upper;
30   return -1;
31 }
```

It's not immediately obvious from looking at this code that it works. So, let's prove that it does, by showing that the precondition implies the loop invariant will be true at the start of the first loop, that if the loop invariant is correct after one iteration of the loop it will be correct after the next iteration, that if the loop terminates and the loop invariants hold, then the postcondition holds, and that the loop does terminate.

Linear search for integer square root

```
1 int isqrt (int n)
2 //@requires n >= 0;
3 //@ensures \result * \result <= n;
4 //@ensures n < (\result+1) * (\result+1) || (\result+1)*(\result+1) < 0;
5 {
6    int i = 0;
7    int k = 0;
8    while (0 <= k && k <= n)
9    //@loop_invariant i * i == k;
10   //@loop_invariant i == 0 || (i > 0 && (i-1)*(i-1) <= n);
11   {
12      // Note: (i + 1)*(i + 1) == i * i + 2*i + 1 and k == i * i
13      k = k + 2*i + 1;
14      i = i + 1;
15   }
16   // This subtraction is necessary since we know k > n now
17   // and i * i == k. i is barely too large to be the square root of n
18   return i - 1;
```

```
19 }
```

Note that this function is very similar to the linear search function we discussed. It's essentially equivalent to searching through an array containing all nonnegative ints less than n, looking for the square root of n.

Binary search for integer square root

Recall the linear search for integer square root function we discussed last recitation. Here's a function that binary searches instead. (Note: this function has a bug related to integer overflow with sufficiently large inputs.)

```
1 #use <util>
2 int bin_search_sqrt (int n)
3 //@requires n > 0;
4 //@ensures \result * \result <= n;
5 //@ensures n < (\result+1) * (\result+1) || (\result+1)*(\result+1) < 0;
6 //@ensures \result == isqrt(n);
7 {
8    int lower = 1;
9    int upper = n;
10   int mid = upper/2;
11   int mid_plus_one_square = (mid + 1) * (mid + 1);
12   while (!(mid * mid <= n
13        && ((mid_plus_one_square > n) || mid_plus_one_square < 0)))
14   // Note that the <= is necessary here because isqrt rounds down.
15   //@loop_invariant lower <= isqrt(n);
16   //@loop_invariant upper >= isqrt(n);
17   {
18      mid = lower + (upper - lower)/2;
19      int square = mid * mid; // Only compute once for efficiency
20      if ((mid != 0 && mid >= int_max() / mid) || square > n) {
21         upper = mid;
22      }
23      else if (square < n) {
24         lower = mid;
25      }
26      else {
27         //@assert mid * mid == n;
28         return mid;
29      }
30      mid_plus_one_square = (mid + 1) * (mid + 1);
31   }
32   return mid;
33 }
```

Note the similarities between this and binary search on an array. If you consider an array `A` of all nonnegative integers, where the ith entry of the array is i^2, we're simply searching for the i such that `A[i] <= n && A[i + 1] > n`. We could implement the function this way, but it would be a large waste of memory.

If we take the square roots of n 5,000,000 times, the UNIX utility `time` reports the following:

n	Binary search time (s)	Linear search time (s)
10,000	1.091	1.413
20,000	1.300	1.934
30,000	1.322	2.374
40,000	1.161	2.863
50,000	1.217	3.187
60,000	1.443	3.447
70,000	1.446	3.777
80,000	1.450	4.032
90,000	1.462	4.292

As you can see, in practice linear search takes much longer than binary search and the amount of time linear search takes increases far more quickly than binary search (as number of elements we're taking the square root of goes up). We'll formalize the notion of linear search being slower than binary search next lecture when we talk about big-O notation.