15-122: Principles of Imperative Computation

Recitation 6 Solutions — 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 fault*s 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.

Binary search

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?

Solution: We're looking for i such that $\frac{n}{2^i} = 1$, or $n = 2^i$. The solution to this, of course, is $\log_2(n) = i$. This gives a rough approximation of how the algorithm's performance changes as the input array size grows. We'll talk more formally about this next week.

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

```
int binsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (-1 == \result && !is_in(x, A, 0, n))
      || ((0 <= \result && \result < n) && A[\result] == x);
 @*/
{
   int lower = 0;
   int upper = n;
   while (lower < upper)
   //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
   //@loop_invariant lower == 0 || A[lower-1] < x;
   //@loop_invariant upper == n || A[upper] > x;
   {
      int mid = lower + (upper-lower)/2;
      if (A[mid] < x) {
         // We can ignore the bottom half of the array now, since we
         // know that every thing in that half must be less than x
         lower = mid+1;
      } else if (A[mid] > x) {
         // We can ignore the upper half of the array, since we know
         // that everything in that half must be greater than x
         upper = mid;
      } else {
         //@assert A[mid] == x;
         return mid;
      }
   }
   //@assert lower == upper;
   return -1;
}
```

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.

Solution:

Precondition implies loop invariant.

Based on the precondition, we know that `0 <= n`. By lines 9 and 10, we know that `0 <= lower` and `upper <= n`. We also know that `lower <= upper`, since `lower == 0` and `upper == n` and `0 <= n`.

Further, `lower == 0` and `upper == n` so the other two loop invariants are true.

Loop invariants are preserved.

Suppose the loop invariants are true at the start of one loop. We now look at one iteration of the loop.

`mid' = lower + (upper - lower) / 2`

We have 3 cases to consider.

Either `A[mid'] < x`, `A[mid'] > x`, or `A[mid'] == x`.

`A[mid'] < x`: In this case, `lower' = mid' + 1`. Since `mid' >= lower` (`(upper - lower)/2` is non-negative), we know that `lower' > lower >= 0`. Since we're in the body of the loop, we know `lower < upper`. Suppose now that `lower' > upper`. Then, `lower + (upper - lower)/2 + 1 > upper`. Rearranging, we get that `2*lower + upper - lower + 2 > 2*upper`. Simplifying, we get `lower + 2 > upper`. Since `lower < upper`, this means that `lower = upper - 1`. However, in that case, `(upper - lower)/2 == 0` because we round, so `lower' == upper`. Thus, `lower <= upper`. `upper' == upper`, so it must still be less than or equal to `n`.

Therefore, the first loop invariant holds.

The second loop invariant must hold because `A[mid'] < x` and `lower' = mid' + 1`. Thus, `A[lower' - 1] == A[mid'] < x`. (Note that `lower' > 0`, since `mid >= 0`).

The third loop invariant must hold because `upper' == upper`.

`A[mid'] > x`: In this case, `upper' == mid'`. `0 <= lower'` since `lower' == lower`.

By the loop guard, `lower < upper`. Since `lower < upper`, `mid' = lower + (upper - lower)/2`, `(upper - lower)/2 > 0`, and `upper' = mid`, we know that `lower == lower' <= mid' == upper'`. Thus, `lower' <= upper'`. Since `mid' <= upper`, `upper' <= upper`. That means that `upper' <= n`.

Finally, `lower` is unchanged, so `0 <= lower` still holds.

The second loop invariant must hold because `lower' == lower`.

The third loop invariant holds because `upper' == mid'` and `A[mid'] > x`. Because `mid' < n`, we can access `A[upper'] == A[mid']`, which is greater than `x` since we were in this case.

`A[mid] == x` In this case, we return immediately, so we never check the loop invariants again.

Thus, the loop invariants hold.

Loop invariants and negation of loop condition imply postcondition.

Next, we show that if the loop invariants hold and we exit the loop, the postcondition holds.

We can exit the loop in two ways: either we enter the `else` branch or `lower >= upper`.

In the first case, we know `A[mid'] == x` since we didn't enter any other case and that this array access is in bounds by the loop invariant, so the postcondition follows.

In the second case, we know `lower >= upper`. So, by the first loop invariant, we know `lower == upper`.

Now we split into cases based on the second and third loop invariants. Either `lower == 0` (and thus `upper == 0`), or `upper == n` (and thus `lower == n`), or neither of those are true and so `A[lower - 1] < x && A[upper] > x`.

Case 1. `A[lower - 1] < x && A[upper] > x`:

We know that `A[lower] > x` since `lower == upper`. Since the array is sorted, `x` would have to be between indices `lower - 1` and `lower`, which isn't possible. Thus, it isn't in the array.

Case 2. `lower == 0` and `upper == 0`:

Thus, either `n == 0` (so `x` isn't in the array), or `A[upper] == A[lower] == A[0] > x` (by loop invariant 3). Since `A` is sorted, this means that `x` is not in `A`.

Case 3. `upper == n && lower == n`:

By the second loop invariant, we know that `n == 0` or `A[n - 1] == A[upper - 1] == A[lower - 1] < x` by the second loop invariant. Since `A` is sorted, that means `x` cannot be in `A`.

Termination. The loop must terminate since the interval between `lower` and `upper` is strictly decreasing in size. If we find the element we're searching for, we return. Otherwise, we eventually get to a point when `lower == upper` and we're done.