15-122: Principles of Imperative Computation

Recitation 5 — 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.

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.)

I claim we can search a sorted array faster than this. We'll discuss why in lecture tomorrow, but for now try to think about how you could improve on this search method.

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. There is a similar improved algorithm that we'll discuss on Friday.