15-122: Principles of Imperative Computation

Recitation 3 — Josh Zimmerman

Arrays

Arrays are stored as *pointers* in C_0. We'll talk more about pointers later in the course, but for now it's sufficient to know that when you have a variable representing an array, it's stored as a reference, not a value. (So, if you pass an array to a function and modify it within the function, the caller of the function will have the modified version.)

```
1  int addOne(int[] arr, int size)
2  //@requires size == \length(arr);
3  {
4     // It's important to make sure that all array accesses are in bounds.
5     // Ideally, we should be able to prove that they are.
6     for (int i = 0; i < size; i++)
7      //@loop_invariant 0 <= i && i <= size;
8      {
9        arr[i] += 1;
10     }
11    // When we take (loop invariant) and not(loop exit condition),
12    // we can easily conclude that i == size, which is often useful
13 }
14
15 int main()
16 {
17    int array_size = 10;
18    int[] a = alloc_array(int, array_size);
19    int[] b = a;
20    //@assert a[0] == 0;
21    //@assert b[0] == 0;
22
23    // We must pass the size of the array to the function because otherwise
24    // the function has no way of verifying that all array accesses are
25    // in bounds.
26    addOne(a, array_size);
27
28    // The function call changes the area in memory that a and b
29    // refer to, so it changes both a and b when we access them.
30    //@assert a[0] == 1;
31    //@assert b[0] == 1;
32 }
```

Here's a slightly more complicated loop: it's a function that calculates the nth Fibonacci number more efficiently than the naive recursive implementation. Assume that we have a function:

```
int slow_fib(int n)
//@requires n >= 0;
;
```

that calculates Fibonacci recursively, and obeys all of the mathematical properties of the Fibonacci sequence. We don't worry about overflow for now – Fibonacci only uses addition, so we can think of it as being defined in terms of modular arithmetic.)

```
1 int fib(int n)
2 //@requires ____________;
3 //@ensures \result == slow_fib(n);
4 {
5   int[] F = alloc_array(int, n);
6   if (n > __) {
7      F[0] = 0;
8   }
9   else {
10     return 0;
11  }
12  if (n > __) {
13     F[1] = 1;
14  }
15  else {
16    return 1;
17  }
18  for (int i = 2; i < n; i++)
19    //@loop_invariant _________________________;
20    //@loop_invariant F[i - 1] == slow_fib(i - 1) && F[i - 2] == slow_fib(i - 2);
21    {
22      F[i] = F[i - 1] + F[i - 2];
23    }
24  return F[n - 1] + F[n - 2];
25 }
```

Fill in the blanks in the code to show that there are no out of bounds array accesses.

Are the invariants strong enough to prove the postcondition?

Nested `for` loops

When looping over a two dimensional array or some data structure (like a picture) that's easiest to think about in two dimensions, nested for loops are very useful. Nested `for` loops are just for loops inside other `for` loops. For an example, see the next page:

```
1 #use <conio>
2 int[][] multiplication_table(int m, int n) {
3    int[][] A = alloc_array(int[], m);
4    for (int i = 0; i < m; i++)
5     //@loop_invariant 0 <= i;
6     {
7       A[i] = alloc_array(int, n);
8       for (int j = 0; j < n; j++)
9        //@loop_invariant 0 <= j;
10       {
11         A[i][j] = i * j;
12       }
13    }
14   return A;
15 }
16
17 int main() {
18   int[][] table = multiplication_table(13, 13);
19   for (int i = 1; i < 13; i++)
20    //@loop_invariant 0 <= i;
21    {
22      for (int j = 1; j < 13; j++)
23       //@loop_invariant 0 <= j;
24       {
25         printint(table[i][j]);
26         print("\t");
27       }
28      print("\n");
29    }
30   return 0;
31 }
```

It's worth noting that you by no means need a two-dimensional array to do this. You can also treat a one-dimensional array as two-dimensional by saying that every n elements you start a new row. You'll be taking advantage of this idea in homework 1.