Upcoming

Quiz: Lectures 1 - 4
There’s going to be a short quiz on the content of the first four lectures. It will be on Blackboard. The professors will email you all when it comes out. Quizzes shouldn’t be scary - they’re a good way to check that you understand some basic ideas. Think of them as sanity checks to gauge how well you’re following along.

Assignment 1: C0, Bitwise Operations, Contracts, Arrays, and Image Manipulation
Each assignment has two parts: theory, and programming. Theory will have short questions, proofs, some code writing. We recommend that you do these with pencil and paper, because your tests will be similar. That said, if we can’t read a solution, we can’t grade it.

Programming will be some *interesting* extension of the concepts discussed in class. In order to accomplish the given task, you’ll have to think critically about the function invariants. You should write your contracts while you’re programming, rather than as an afterthought. They are excellent for debugging.

Bit Operations

<table>
<thead>
<tr>
<th>z</th>
<th>$z_i$ (the $i^{th}$ bit of $z$)</th>
<th>Examples (write here)</th>
<th>z</th>
<th>$z$ (what happens to $x$)</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &amp; y$</td>
<td>1, If $x_i = 1$ and $y_i = 1$ 0, If $x_i = 0$ or $y_i = 0$</td>
<td></td>
<td>$x &lt;&lt; y$</td>
<td>Shift the bits of $x$ left $(y \text{ mod } 32)$ places</td>
<td></td>
</tr>
<tr>
<td>$x \mid y$</td>
<td>1, If $x_i = 1$ or $y_i = 1$ 0, If $x_i = 0$ and $y_i = 0$</td>
<td></td>
<td>$x &gt;&gt; y$</td>
<td>Shift the bits of $x$ right $(y \text{ mod } 32)$ places with sign extension*</td>
<td></td>
</tr>
<tr>
<td>$x ^ y$</td>
<td>1, If $x_i \neq y_i$ 0, If $x_i = y_i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\sim x$</td>
<td>1, If $x_i = 0$ 0, If $x_i = 1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Sign extension means that when you’re shifting the most significant bit, you copy it as you go along. So if you’re working with 4-bit integers, $1110 >> 1 = 1111$ and $0111 >> 1 = 0011$.

Pixels as ARGB Ints

ARGB pixels present a fantastic example of where we care more about the hexadecimal (and binary) representation of a number more than its decimal representation. Consider a pixel $p$. $p_{10} = 424362039$ tells us nothing. On the other hand, $p_{2} = 111111001110000100000110001111$ tells us quite a bit, and $p_{16} = 0xFCF0818F$ tells us even more.

An ARGB pixel value can be interpreted as follows:

<table>
<thead>
<tr>
<th>Alpha</th>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{16}$</td>
<td>F</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>$p_{2}$</td>
<td>1111</td>
<td>1100</td>
<td>1111</td>
</tr>
</tbody>
</table>

How would you return the magnitude of the green channel in $p$?
How would you make $p$ entirely opaque?
How would you invert the blue channel?

**Arrays**

Arrays are sequences of data of a particular type that can be accessed by their index, which starts at 0.
For example, let `int[] F` be the following array: [1 1 2 3 5 8 13]
Then $F[0]$ is ____, $F[2]$ is ____, and $F[4]$ is ____.

How would we go about constructing $F$? Here is a function to do that.

```c
/* compute the first n fibonacci numbers and return them in an array */
int[] fib(int n)
//@requires 0 <= n;
//@ensures \length(\result) == n;
{
    int[] F = alloc_array(int, n);
    if (n > 0) F[0] = 0;
    if (n > 1) F[1] = 1;
    for (int i = 2; i < n; i++)
//@loop_invariant 2 <= i;
    {
        F[i] = F[i-1] + F[i-2];
    }
    return F;
}
```

Some important properties of arrays:

1. There is no function in $C_0$ that will tell you the length of the array. $\\text{length}(arr)$ can only be used in contracts.
2. If you have an array of length $n$, valid array accesses are only those in $[0, n)$

**Safe array access**

You always want to be able to show that you aren’t accessing your array at an invalid index.

Why is it okay to talk about $F[0]$ in `fib`? $F[1]$? It’s because we only access them if we know that the array is big enough.

Additionally, loop invariants can tell us that our accesses are safe when we’re traversing an array in a loop. Notice that the loop invariant in `fib` is $2 <= i$. We know it’s true before entering the loop because $i$ is initialized to 2. It is not always true that $i <= n$, though, since we’ve just handled cases where the array has length 1 or 0. But that’s okay, because if the array is too small, we won’t enter the loop, so it is a strong enough statement to just talk about the lower bound of $i$ in this case. If we do enter the loop, though, we can use the fact that $2 <= i$ and $i < n$ to justify access of $F[i]$, $F[i-1]$, and $F[i-2]$.

**Aliasing**

When you set one array equal to another, you alias it. If two or more variables are aliases of each other, then they point to the exact same space in memory. If you use one of those variables to modify the array, then you’ve modified the array for all of the variables, like so:

```c
int[] F = fib(15);
int[] G = F;
G[5] = 7;
//@assert F[5] == 7;
```
However, if you have two variables aliased to the same array, and you reassign one of them, you’ve only reassigned that one, making the variables no longer aliases. For example:

```plaintext
int[] F = fib(15);
int[] G = F;
int[] G = fib(15);
G[5] = 7;
//@assert F[5] == 5;
```

Aliasing is not a problem when you’re dealing with literal values like ints, since the variable contains their value rather than points to where the value is stored. For example:

```plaintext
int[] F = fib(15);
int x = F[3];
//@assert x == 2;
F[3] = 4;
//@assert x == 2;
```

**Loop Invariants beyond ensuring safe access**

We didn’t get to this in recitation, but here’s another interesting use of loop invariants to make statements about your array as you’re traversing and modifying it. Consider the following functions:

```plaintext
/* checks that the A[0] through A[n-1] are true */
bool all_true(bool[] A, int n)
//@requires 0 <= n && n <= \length(A);
{
   for (int i = 0; i < n; i++)
      //@loop_invariant 0 <= i && i <= n;
      { if (!A[i]) return false; }
   return true;
}

/* sets A[0] through A[n-1] to true */
void set_all_true (bool[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures all_true(A, n);
{
   for(int i = 0; i < n; i++)
      //@loop_invariant 0 <= i && i <= n;
      //@loop_invariant all_true(A, i);
      { A[i] = true; }
   return;
}
```

On its own, `all_true` isn’t that interesting. But the way it’s used in `set_all_true` gives us exactly what we need to prove the post-condition. Notice that `all_true(A, i)` tells you that all the elements in `A` from 0 through `i` are `true`, and that when we exit the loop (when `i == n`,) we know that `all_true(A, n)`, which is just what we want.