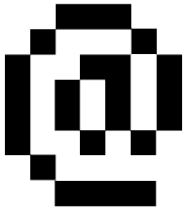
## 15-122: Principles of Imperative Computation, Spring 2013

## Programming 1: Pixels



Due: Monday, September 9, 2013 by 22:00

This second programming assignment is designed to get you used to writing some preconditions and postconditions, and also deals with operations on integers.

The code handout for this assignment is at

`http://www.cs.cmu.edu/~rjsimmon/15122-f13/hw/15122-prog1.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is no limit on the number of times you may hand in this assignment on Autolab.

# 1 Pixels

To capture the contents of a single pixel, we need to know two things: how opaque or transparent it is, and what color it is.

One common way to do this is called *ARGB*.[1] The transparency is stored as an integer in the range $[0, 256)$, where 0 is completely transparent and 255 is completely opaque. This is called the *alpha (A)* value. The color is stored as three other integers, each also in the range $[0, 256)$, which respectively describe the intensity of the *red (R)*, *green (G)*, and *blue (B)* color in the pixel.

So a pixel is described by four integers between 0 (inclusive) and 256 (exclusive). One way to *represent* the four integers that make up a pixel is by packing them inside a 32-bit C0 integer, breaking that integer up into 4 components with 8 bits each:

$$a_0a_1a_2a_3a_4a_5a_6a_7\, r_0r_1r_2r_3r_4r_5r_6r_7\, g_0g_1g_2g_3g_4g_5g_6g_7\, b_0b_1b_2b_3b_4b_5b_6b_7$$

where:

| | |
|---|---|
| $a_0a_1a_2a_3a_4a_5a_6a_7$ | represents the alpha value (how opaque the pixel is) |
| $r_0r_1r_2r_3r_4r_5r_6r_7$ | represents the intensity of the red component of the pixel |
| $g_0g_1g_2g_3g_4g_5g_6g_7$ | represents the intensity of the green component of the pixel |
| $b_0b_1b_2b_3b_4b_5b_6b_7$ | represents the intensity of the blue component of the pixel |

Each 8-bit component can range between a minimum of 0 (binary 00000000 or hex 0x00) to a maximum of 255 (binary 11111111 or hex 0xFF).

[1] `http://en.wikipedia.org/wiki/RGBA_color_space`

In the file `pixel.c0`, right at the top we announce that we will be working with a type `pixel` that is actually represented as a single integer by writing a *type definition*:

```
typedef int pixel;
```

The rest of the file should contain the implementation of an interface to the newly-defined pixel type. By using this interface, we can manipulate pixels as four integers for red, green, blue, and alpha values instead of worrying exactly how they are packed into an integer.

**TASK 1** ( 3 pts.) Complete the C0 file `pixel.c0`. Translate the English descriptions into code and the English contracts into C0 contracts.

You can load your completed file into `coin`. Remember to use the `-d` flag to check contracts.

```
% coin -d pixel.c0
--> make_pixel(255, 238, 127, 45);
```

# 2 Testing

We can generally think about three ways that a program might fail:

1. Do something *unsafe*: access an array out of bounds, divide by zero, call a function with inputs that violate the function's preconditions.
2. Violate a loop invariant, an assertion, or a postcondition.
3. Return the wrong answer without violating any contracts.
4. Fail to terminate.

For the fast exponent function we considered in lectures 1 and 2, failure #3 was impossible: the postcondition specified that exactly the right answer was returned. That won't always be the case, and it wasn't the case for `pixel.c0`.

**TASK 2** ( 2 pts.) Make a copy of the `pixel.c0` file named `pixel-bad.c0`:

```
% cp pixel.c0 pixel-bad.c0
```

Edit this file so that it contains a broken implementation of pixels. Keep the contracts the same, and avoid failures #1 and #4 – the program should remain safe and should terminate. However, at least one function should sometimes violate its postcondition (#2, a contract failure) and at least one function should sometimes give the wrong answer without violating a postcondition (#3, a contract exploit).

**TASK 3** ( 3 pts.) Write a file `pixel-test.c0` that checks for both contract failures and contract exploits in an implementation of the pixels interface. (See the notes from Recitation 2 for an example of how to do this for the greatest common divisor function.) At minimum, the test should catch the bugs you made intentionally:

```
% cc0 -w -d pixel.c0 pixel-test.c0
% ./a.out

   <Should run without errors>

% cc0 -w -d pixel-bad.c0 pixel-test.c0
% ./a.out

   <An assertion should fail>
```

On Autolab we'll run your tests against some of our buggy pixel implementations too; you'll need to catch bugs in our buggy pixel implementations for full credit.

# 3 Pixel manipulation and array aliasing

The comments in `hw1.c0` walk through the tasks in the rest of the assignment: red removal, quantization, and division-by-subtraction. You can run and test these programs with `coin`:

```
% coin -d pixel.c0 hw1.c0
```

or you can write, compile, and run a test file like `hw1-test.c0`:

```
% cc0 -d pixel.c0 hw1.c0 hw1-test.c0
% ./a.out
```

**TASKS 4-7** ( 7 pts.) Complete the tasks described in `hw1.c0`.

Task 6 asks you to implement *quantization*, a transformation on pixels. Quantization can be performed on all the pixels in an image to reduce the total number of colors used in that image.

Given a pixel and a quantization level $q$ in the range $[0, 8)$, we quantize by taking each color component (red, green and blue) and clearing the lowest $q$ bits. For example, suppose we have a pixel with red intensity $R = 107$, green intensity $G = 190$, and blue intensity $B = 215$. The color components of this pixel are represented by these bytes:

```
RED       GREEN     BLUE
01101011  10111110  11010111
```

If the quantization level is 5, then the resulting pixel should have the following color components (note how the lower 5 bits are all cleared to 0):

```
RED       GREEN     BLUE
01100000  10100000  11000000
```

A pixel processed with a quantization level of 0 should not change. For each pixel, do not change its alpha component.