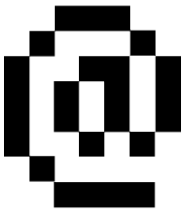
15-122: Principles of Imperative Computation, Spring 2016

Programming 2: Pixels



Due: Thursday 28th January, 2016 by 22:00

This second programming assignment is designed to get you used to writing some preconditions and postconditions, deals with operations on integers, and introduces the idea of an interface.

The code handout for this assignment is on Autolab and at

`http://www.qatar.cmu.edu/~srazak/courses/15122-s16/hws/pixels-handout.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 numbers between 0 (inclusive) and 256 (exclusive).

There are many ways to *represent* a pixel! One way to take the four numbers that make up a pixel is by packing them inside a 32-bit C0 **`int`**, breaking that **`int`** up into 4 components with 8 bits each:

$$a_0a_1a_2a_3a_4a_5a_6a_7\ r_0r_1r_2r_3r_4r_5r_6r_7\ g_0g_1g_2g_3g_4g_5g_6g_7\ b_0b_1b_2b_3b_4b_5b_6b_7$$

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

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`).

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 (see Section 3 for what interfaces are exactly). 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 points) 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 four 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 points) 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 points) Write a file `pixel-test.c0` that checks for both contract failures and contract exploits in an implementation of the pixels interface. (See Appendix A or the file `puzzle-test.c0` distributed with the previous programming homework for an example of how to do this for the `common_prefix` 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 Introduction to interfaces

It's useful to be able to store all the parts of a pixel within a single integer. But it's not necessary to store the alpha value in the leftmost (i.e. *high-order*) 8 bits, nor is it necessary to store the blue value in the rightmost (i.e. *low-order*) 8 bits. In fact, it's not even necessary to store pixels as integers at all! The file `pixel.c0` defines the type `pixel` and defines five functions: `make_pixel(a,r,g,b)` tells us how we can create pixels, and `get_red(p)`, `get_green(p)`, `get_blue(p)`, and `get_alpha(p)` tell us what we can do to pixels. We can say that these five functions form the *interface* to pixels — if we only use those five functions to interact with the `pixel` type, then we can easily change the representation of pixels without any of our code breaking. It's the *implementation* you wrote that declares a `pixel` to be an 32-bit integer.

A simple way we might change the implementation would be to store the bits in a different order. A more drastic way that we might change the implementation is in the file `pixel-array.c0`. In that implementation, pixels are stored not as single integers but as arrays of four integers:

```
% coin -d pixel.c0
--> pixel p = make_pixel(255,238,127,45);
p is -1147091 (int)
--> get_green(p);
127 (int)

% coin -d pixel-array.c0
--> pixel p = make_pixel(255,238,127,45);
p is 0x603A60 (int[] with 4 elements)
--> get_green(p);
127 (int)
```

While the person implementing the pixel interface obviously knows whether a pixel is an integer or an array, the person using the pixel interface should treat the type pixel as an unknown type (or *abstract type*), and shouldn't rely on details of how the type is implemented. In this class, we'll use a typedef with underscores to emphasize that an interface defines an abstract type:

```
typedef ______ pixel;
```

This notation isn't actual valid C0, though, so you'll often see it as a comment in a C0 file next to the actual type definition.

An interface allows us to separate the *library code*, which understands the implementation details, from the *client code*, which only knows about the interface. Setting up interfaces is an important part of writing code — and this is even true when you're the person writing both the library code and the client code! Interfaces are the basis of how we organize our code and our large software projects. We'll be talking more about interfaces later in this class.

4 Pixel manipulation and array aliasing

In this last part of this assignment, you will write code that uses this `pixel` interface:

```
/* Interface to pixels */

typedef ______ pixel

pixel make_pixel(int alpha, int red, int green, int blue)
int get_alpha(pixel p)
int get_red(pixel p)
int get_green(pixel p)
int get_blue(pixel p)
```

The code you write for these tasks should *respect the pixel interface* — that is, your code shouldn't make any assumptions about what a pixel is other than that a pixel can be created with the `make_pixel` function and passed to the four `get_` functions. If you write code that respects the pixel interface, then you should be able to test your `tasks.c0` file against both your `pixel.c0` and `pixel-array.c0`.

```
% coin -d pixel.c0 tasks.c0
% coin -d pixel-array.c0 tasks.c0
```

The converse is nearly true as well: if your `tasks.c0` can compile and run against both `pixel.c0` and `pixel-array.c0`, you can be pretty confident that it respects the interface.

The comments in `tasks.c0` walk through the tasks in the rest of the assignment: red removal, quantization, and returning multiple arguments. You can run and test your code with `coin` as described above, or you can write, compile, and run a test file like `tasks-test.c0`, as described in `README.txt`.

Task 4 (1 point) Complete function `remove_red` in file `tasks.c0`.

Task 5 (2 points) Complete function `quantize` in file `tasks.c0`.

Quantization is a transformation on pixels. It 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.

Task 6 (2 points) Complete function `test_quantize` in file `tasks.c0`.

Task 7 (2 points) Complete function `count_zeroes` in file `tasks.c0`.

A Testing GCD

Say we have a function that is supposed to find the *greatest common divisor* of two positive integers. (We haven't talked about how to write such a function, but you've seen bits and pieces; search for "Euclid's algorithm" if you'd like to implement this function.)

```
int gcd(int x, int y)
//@requires x > 0 && y > 0;
//@ensures 0 < \result && x % \result == 0 && y % \result == 0;
```

The postcondition isn't the best one we could write — it checks that the result is *a* divisor of `x` and `y`, not *the greatest* common divisor. A function that ignores its inputs and always returns 1 satisfies this contract but is nevertheless an incorrect implementation of `gcd`.

We'll write some unit tests in a file `gcd-test.c0` that includes a `main` function. To check for contract exploits, we need to make extra assertions that the output of the function is correct. We could do this with the contract, but it also makes sense to use the built-in `assert()` function that runs whether or not `-d` is selected.

```
#use <util>
#use <conio>

int main() {
  // Run some edge cases (check for contract errors only)
  gcd(1, 1);
  gcd(1, int_max());
  gcd(int_max(), int_max());
  gcd(int_max(), int_max() - 1);

  // Test some regular cases (check for contract errors & exploits)
  assert(gcd(2, 5) == 1);
  assert(gcd(19, 21) == 1);
  assert(gcd(81, 9) == 9);
  assert(gcd(16, 100) == 4);

  println("All tests passed!");
  return 0;
}
```

Now we can use this test file to test both good and bad implementations of GCD:

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

```
All tests passed!
0
% cc0 -w -d gcd-bad.c0 gcd-test.c0
% ./a.out
gcd-test.c0:14.3-14.27: assert failed
Abort trap: 6
```