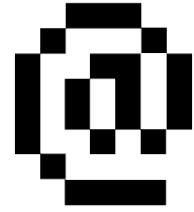# 15-122: Principles of Imperative Computation, Spring 2023

Programming Homework 2: Pixels

Due: Wednesday 25$^{\text{th}}$ January, 2023 by 9pm

This second programming assignment is designed to get you used to writing some preconditions and postconditions, deals with bitwise operations on integers, and introduces the constituent parts of a library: the interface and the implementation.

Download the assignment handout from the course website or Autolab. 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 a EIGHT (8) PENALTY-FREE HANDIN LIMIT, with the idea that for each task you can test your code, hand in, and then fix any bugs found by Autolab while working on and testing the next task. Every additional handin will incur a small (5%) penalty (even if using a late day). Your score for this assignment will be the score of your last Autolab submission.

1   A Pixel Library

In the first part of this assignment, we will write a few functions that manipulate pixels.

To describe an individual pixel, we need to know two things: how opaque or transparent it is, and what color it is. The transparency is an integer in the range $[0, 256)$, where 0 means completely transparent and 255 means completely opaque. This is called the *alpha* value of the pixel. The color of the pixel is expressed by three other integers, each also in the range $[0, 256)$, which respectively describe the intensity of the *red*, *green*, and *blue* color in the pixel. This way of describing a pixel is called the ARGB color model[1] from the first letter of these four values. So, in the ARGB model, a pixel is described by four numbers between 0 and 255 (both inclusive).

To manipulate pixels, all we need are an operation that manufactures a pixel given values for its alpha, red, green, and blue components, and four operations that return the alpha, the red, the green, and the blue component of a given pixel, respectively.

A **pixel library** provides a *type* so that we can refer to and use pixels in our programs, and *functions* that implement these five operations. Here is what we need to know to use the pixel library:

```
// typedef _____ pixel_t;                      // Type for pixels

// Returns a pixel_t representing an ARGB pixel consisting of the given
// alpha, red, green and blue intensity values.
pixel_t make_pixel(int alpha, int red, int green, int blue)
/* PRECONDITION: all intensity values must be between 0 and 255,
                 inclusive. */ ;

// Returns the alpha component of the given pixel p.
int get_alpha(pixel_t p)
/* POSTCONDITION: returned value is between 0 and 255, inclusive */ ;

// Returns the red component of the given pixel p.
int get_red(pixel_t p)
/* POSTCONDITION: returned value is between 0 and 255, inclusive */ ;

// Returns the green component of the given pixel p.
int get_green(pixel_t p)
/* POSTCONDITION: returned value is between 0 and 255, inclusive */ ;

// Returns the blue component of the given pixel p.
int get_blue(pixel_t p)
/* POSTCONDITION: returned value is between 0 and 255, inclusive */ ;

// Prints the given pixel p to terminal.
void pixel_print(pixel_t p);
```

---

[1]http://en.wikipedia.org/wiki/RGBA_color_model

This is called the ***interface*** of the library. It tells us that the type of pixels is called `pixel_t` — this allows us to declare variables of type `pixel_t` so that we can work with pixels in our programs. The interface also lists the functions the library provides to manipulate pixels. It may appear funny that pre- and post-conditions are written as comments (in C0!). We will address this shortly.

But what exactly is a `pixel_t`?

It turns out that we don't need to know! We can write lots of useful programs that manipulate pixels by just using the type `pixel_t` and the above five functions. You don't believe us? Let's give it a try.

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 the pixel 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 = $ `0x6B` (decimal 107, binary `01101011`), green intensity $G = $ `0xBE` (decimal 190, binary `10111110`), and blue intensity $B = $ `0xD7` (decimal 215, binary `11010111`). 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. Quantization does not change the alpha component of a pixel.

Task 1 (3 points) Complete function `quantize` in file `quantize.c0`.

You can test your code in `coin` by running the command

```
% coin -d pixel.o0 quantize.c0
```

There, you can make a few pixels, call `quantize` with different quantization levels on them, and then check that the various components of the quantized pixels are what you expect.

The *object file* `pixel.o0` is a binary version of one of the many possible implementations of the pixel library. Being a binary file, it hides the definition of the type `pixel_t` and the implementation of the five functions on pixels listed in the interface.

Running tests in `coin` gets tedious after a while. You are much better off writing them down in a way that makes it easy to run them over and over. This is what we'll do next.

Task 2 (3 points) Complete the function `test_quantize` in file `quantize.c0`.

Now, a faster way to test your code is to compile and run it with

```
% cc0 -d -W -o quantize-test1 pixel.o0 quantize.c0 quantize-test.c0
% ./quantize-test1
```

2 Implementing a Pixel Library

Libraries are a big deal in Computer Science. We will see a lot of libraries later in this course, and we will implement many of them. As a preview (and as a way to practice several concepts we saw so far in the course), we will consider a new ***implementation*** of the pixel library, different from the one in `pixel.o0`.

The implementation you will develop in this assignment represents a pixel as a single C0 **int**. Since the value of each component of a pixel ranges from 0 to 255, eight bits are sufficient to represent a component, and $4 \times 8 = 32$ is the number of bits in an **int**! We pack the four intensities in a 32-bit **int** as follows:

$$a_0a_1a_2a_3a_4a_5a_6a_7 \; r_0r_1r_2r_3r_4r_5r_6r_7 \; g_0g_1g_2g_3g_4g_5g_6g_7 \; b_0b_1b_2b_3b_4b_5b_6b_7$$

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`).

But now that we have a new representation of pixels, we need to write the five functions that operate on them (that's `make_pixel` and the four `get_...` functions). We will do so in the file `pixel-int.c0`. This file contains the ***implementation*** of this new pixel library. It has the same interface as `pixel.o0` but a totally different code for the type `pixel_t` and the pixel operations.

Let's look at this file. It starts with the line

```
typedef int pixel;        // Library (concrete) view of a pixel
```

This defines the type `pixel` which the library implementation will use to refer to pixels internally: it says that `pixel` is the same thing as **int**. (We could use **int** everywhere, but we would get confused as **int** would stand both for a pixel and for its components.)

The file `pixel-int.c0` continues with the implementation of the pixel operations (which currently all have an empty body except `pixel_print`).

This file ends with the line

```
typedef pixel pixel_t;   // Client (abstract) view of a pixel
```

which defines the type `pixel_t` mentioned in the interface as simply `pixel`. Why so many types? This library is tiny and there would be little harm in using `pixel_t` throughout. Libraries we will see later in this course are more complex, and keeping the type mentioned in the interface (here `pixel_t`) distinct from the type used in the implementation (here `pixel`) will have a number of advantages.

Let's implement `pixel-int.c0`!

Task 3 (4 points)   Complete the C0 file `pixel-int.c0`. This has two parts:

- Translate the English contracts into C0 contracts.
- Translate the English description of the pixel functions into code. You may only use bitwise operations and shifts.

The first part (the contracts) and half of the second part (the code) will be graded *manually*. Because of this, 2.5 points will show up a couple of days after the submission deadline. You do not need to do anything for `pixel_print`.

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

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

In future libraries, we will write some contracts directly in the interface. Stay tuned!

3   Respecting the Interface

When using a library in an application, you should only use the types and functions mentioned in its interface. In particular, your application should not rely on the details of how the library was implemented. This is called *respecting the interface*.

But why? Conceptually, a pixel is its own thing and the interface functions allow manipulating it as such — the interface provides us with a welcome *abstraction* of what a pixel is. Practically, if you only use what is provided by the interface, we can swap the library with a different one with the same interface (for example `pixel.o0` with `pixel-int.c0`) and everything will work the same in your application.

Did the code you wrote in tasks 1–2 respect the interface? Carefully check that you only used the type and functions mentioned in the interface! Then, try running

```
% cc0 -d -W -o quantize-test2 pixel-int.c0 quantize.c0 quantize-test.c0
```

(Notice that we swapped `pixel.o0` with `pixel-int.c0`.) Your earlier code should still compile with no errors. Furthermore, running `./quantize-test2` should produce the exact same output as `./quantize-test1`.

Task 4 (2 points)   If the above command did not result in compilation errors and `./quantize-test2` worked just like `./quantize-test1`, you may not need to do anything in this task. Otherwise, you need to update the code you wrote in `quantize.c0` (and possibly `quantize-test.c0`) so that it respects the interface.

Hopefully, you wrote interface-respecting code from the get-go. Just in case, the next task asks you to spot and fix code that doesn't.

Task 5 (2 points)   Update the function `remove_red` in file `respect.c0` so that it respects the interface of the pixel library. See `README.txt` for how to compile and run your code.

In this course, we will (nearly) always write code that respects the interface.

4   Testing

Let's make a new implementation of the pixel library, but this time a broken one.
   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 mystery 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 isn't the case for the pixel library.

Task 6 (2 points)   Make a copy of the `pixel-int.c0` file named `pixel-bad.c0`:

```
% cp pixel-int.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*). Since you may make arbitrary changes in `pixel-bad.c0`, we will test its `get_...` functions only on pixels returned by its `make_pixel`.

Task 7 (7 points)   The course staff wrote a number of buggy implementations of the pixel library on Autolab. In this task, your job will be to write test cases that expose our bugs.
   In file `pixel-test.c0`, write tests that check for both contract failures and contract exploits in an implementation of the pixels interface. (See Appendix A, or the previous programming homework for how to write tests.) At a minimum, your tests should run without errors for a correct implementation of the pixel interface, and catch the bugs you made intentionally in the previous task:

```
% cc0 -d pixel-int.c0 pixel-test.c0
% ./a.out
   <Should run without errors>

% cc0 -d pixel-bad.c0 pixel-test.c0
% ./a.out
   <An assertion should fail>
```

5 Arrays of Pixels

As you will see in the *next* programming assignment, an image is essentially an array of pixels. And arrays can play tricks on you!

Task 8 (2 points) File `multireturn.c0` contains the buggy function `count_zeroes`, some correct helper functions, and some tests for `count_zeroes`. You can compile and run it as follows:

```
% cc0 -d -W pixel.o0 multireturn.c0 multireturn-test.c0
% ./a.out
```

Identify the bug(s) in `count_zeroes` by using some of the debugging techniques seen in class, such as tracing execution on a C0 memory diagram, or printing out the result array and comparing it with what you expected. Write more tests! Do not make assumptions about the inputs of `count_zeroes` other than what the preconditions say.

Once you have identified what is wrong, fix the code and test your fixes. Make sure to respect the pixel interface!

If you get lost as you make changes to `multireturn.c0` and want to go back to the original version, you can copy it from `multireturn-backup.c0`.

A Appendix: Testing GCD

This appendix demonstrates how to write a test file with an example function.

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 their *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 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 `@assert` contract, but it's better to use the built-in `assert()` function that runs whether or not `-d` is selected.

```
1 #use <util>
2 #use <conio>
3
4 int main() {
5   // Test some regular cases
6   assert(gcd(2, 5) == 1);
7   assert(gcd(19, 21) == 1);
8   assert(gcd(81, 9) == 9);
9   assert(gcd(16, 100) == 4);
```

```
10
11    // Test some edge cases
12    assert(gcd(1, 1) == 1);
13    assert(gcd(1, int_max()) == 1);                      // This may take a while
14    assert(gcd(int_max(), int_max()) == int_max());  // This too
15    assert(gcd(int_max(), int_max() - 1) == 1);
16
17    printf("All tests passed!\n");
18    return 0;
19 }
```

Edge cases are inputs at the range boundaries, for example smallest and largest inputs. In this case, the precondition requires that both of the inputs are positive, so the smallest they can be is 1 and the largest is int_max().

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

```
% cc0 -d gcd.c0 gcd-test.c0
% ./a.out
All tests passed!
0
% cc0 -d gcd-bad.c0 gcd-test.c0
% ./a.out
c0rt: gcd-test.c0: 12.6-12.29: assert failed
at main (gcd-test.c0: 12.6-12.29)
Aborted (core dumped)
```