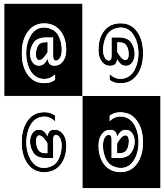
15-122: Principles of Imperative Computation, Fall 2022

Programming Homework 3: Images



Due: Thursday 22nd September, 2022 by 9pm EDT

This programming assignment will have you using arrays to represent and manipulate images.

From your `private/15122` directory, run

```
% autolab122 download images
```

to download the code handout for this assignment (you can also find it on 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 SIX (6) 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. Make sure to leave enough submissions to work on the optional Task 5, if you wish to do that. 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.

Style Grading: With this assignment, we will begin to emphasize *programming style* more heavily. We will actually be looking at your code and evaluating it based on the criteria outlined in the "Coding with Style" Guide to Success on Diderot. We will make comments on your code via Autolab, and will assign an overall passing or failing style grade. A failing style grade will be temporarily represented as a score of -15 points. This -15 will be reset to 0 once you:

1. fix the style issues,
2. see a member of the course staff during office hours **within 5 days** after the grades are released, and
3. briefly discuss the style issues and how they were addressed.

We will evaluate your code for style in two ways. We will use `cc0` with the `-w` flag that gives style warnings — code that raises warnings with this flag is almost certain to fail style grading. Because the `-w` flag does not check for good variable names, appropriate comments, or appropriate use of the functions defined in `pixel.o0` and `imageutil.c0`, these issues will be checked by hand.

Task 1 (3 points) In addition to using good style, be sure to include appropriate contracts, `@requires`, `@ensures`, and `@loop_invariant`. Your annotations should be sufficient to ensure that all array accesses in your functions are safe and that whoever calls your functions has sufficient information to make safe array accesses. The points for this task will be assigned based on a visual inspection of the contracts you write throughout your code for this assignment.

1 Image manipulation

This assignment uses a pixel library similar to what you implemented in the last homework. You can find its interface in Appendix A.

The programming problems you have for this assignment deal with manipulating images. An image will be stored in a one-dimensional array of pixels. (The C0 `<img>` image library assumes the ARGB implementation of pixels that you wrote in your last assignment.) Pixels are stored in the array row by row, left to right starting at the top left of the image. For example, if a 5×5 image has the following pixel "values":

$$\begin{array}{ccccc} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \\ u & v & w & x & y \end{array}$$

then these values would be stored in the array in this order:

$$a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p\ q\ r\ s\ t\ u\ v\ w\ x\ y$$

In the 5×5 image, the pixel i is in row 1, column 3 (rows and columns are indexed starting with 0) but is stored in the one-dimensional array at index 8. An image must have at least one pixel.

Task 2 (4 points) Complete the C0 file `imageutil.c0`. As with the `pixel-int.c0` implementation from your last assignment, you must fill in the missing code and translate the English preconditions and postconditions into `@requires` and `@ensures` statements.

We do not require you to hand in the file `images-test.c0` where you can write tests for your *imageutil* implementation the way you tested your *pixels* implementation. It is a good idea to write a lot of tests, however! (Recall also that you have a limited number of submissions.)

2 Image Transformations

The rest of this assignment involves implementing the core part of a series of image transformations. Each function you write will take an array representation of the input image(s) and return an array representation of the output image. These functions should *not* be destructive: you should make your changes in a copy of the array, and not make any changes to the original array. Your implementations should be relatively efficient, meaning both that they should have a reasonable big-O running time and that they should take at most a few seconds to run on our example images.

Remember that your code should have appropriate preconditions and postconditions. It is always a precondition that the given width and height are a valid image size that matches the length of the pixel array passed to the function. It is always a postcondition that the returned array is a different array from any array that was passed in, and that this resulting array has the correct length.

In order to pass style grading, you will be expected to use functions from the *pixel* interface (the type `pixel_t` and functions `get_red`, `get_green`, `get_blue`, `get_alpha`, and `make_pixel`) and the *imageutil* interface (the functions `is_valid_imagesize`, `get_row`, `get_column`, `is_valid_pixel`, `get_index`) in the next two tasks. On Autolab, we will compile your code for the remaining tasks against *our* implementation of the *pixel* and the *imageutil* interfaces, so you cannot add new functions to these interfaces.

Testing. There are two ways to test your code. We encourage you to use both.

1. Create tiny images manually in file `images-test.c0`, call your functions on them, and then print the results using the provided `image_print` function. This is most useful on images smaller than, say, 10×10 pixels.

2. Once your code works well on small images, use the provided `*-main.c0` files to test it on larger images. How to do so is described in the `README.txt` in the code handout.

   We are providing a program, `imagediff`, to help you compare your output images to the sample images in the handout, optionally saving an image that shows you exactly where the two images differ. It is in the course directory on `afs`, so it is available on any cluster machine or when you are connected via ssh. For example:

   ```
   % imagediff -i img/sample.png -j img/my-image.png -o img/diff.png
   ```

   This command compares the image `img/sample.png` and `img/my-image.png` and creates a visual representation of the difference in `img/diff.png`.

Figure 1: A sporty coupe before and after blue removal.

2.1 Removing blue

As an example of image manipulation, you should take a look at `remove-blue.c0`. The core of this transformation is this function:

```
pixel_t[] remove_blue(pixel_t[] pixels, int width, int height)
```

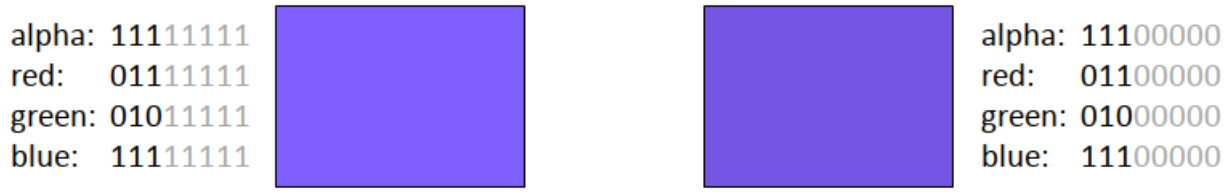
An example of this transformation is given in Figure 1.

You should look at the file `remove-blue.c0` to get an idea of how this transformation works, and you should look at `README.txt` to see how to compile and run this transformation against `remove-blue-main.c0`. You are strongly encouraged to write some smaller test cases for your programs. An example of what this should look like is given in `remove-blue-test.c0`.

Note that `remove-blue.c0` doesn't use the *pixel* or *imageutil* libraries. If *your* code doesn't use the *pixel* or *imageutil* libraries, you will fail style grading! While it is not required, you might want to try your hand at modifying `remove-blue.c0` to use the *pixel* and *imageutil* libraries.

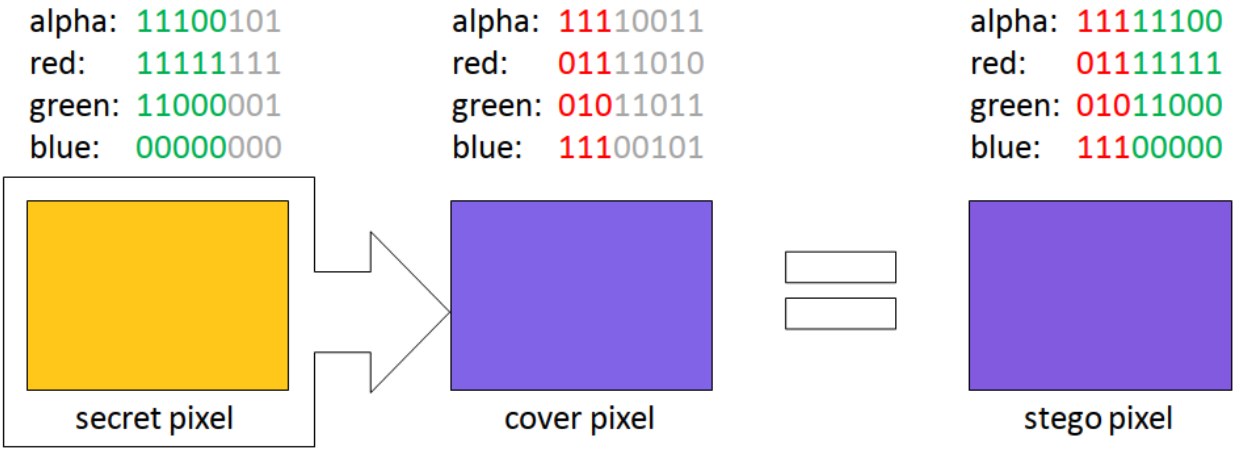
2.2 Hiding an Image inside Another

Given any pixel, the most significant (i.e., leftmost) bit of every channel contributes 50% of the color (and transparency) of the pixel. The next bit (the second leftmost) contributes 25%, the next after that contributes 12.5%, and so on up to the least significant (rightmost) bit which contributes a mere 0.4% of the appearance of the pixel. So, for example, the left three bits of each channel contribute 50% + 25% + 12.5% = 87.5% of the appearance of a pixel. This can be seen in the following figure: the magnified pixel on the left has its rightmost five bits set to 1 and the pixel on the right has them set to 0.



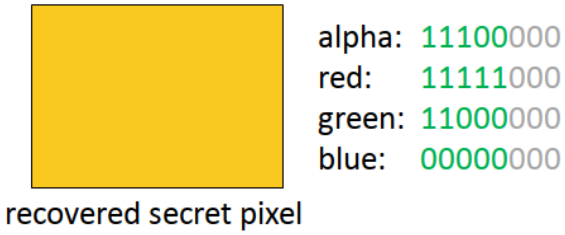
Observe that, to the naked eye, these two pixels look almost identical. The same would hold true no matter what value had we used for the five least significant bits of each channel.

We can exploit this observation to hide a (secret) pixel in another (the cover pixel). The idea is to replace the five least significant bits of each channel of the cover pixel with the five most significant bits of the corresponding channel of the secret pixel. Here's an example:



The resulting pixel, the stego pixel, is nearly indistinguishable from the cover pixel.

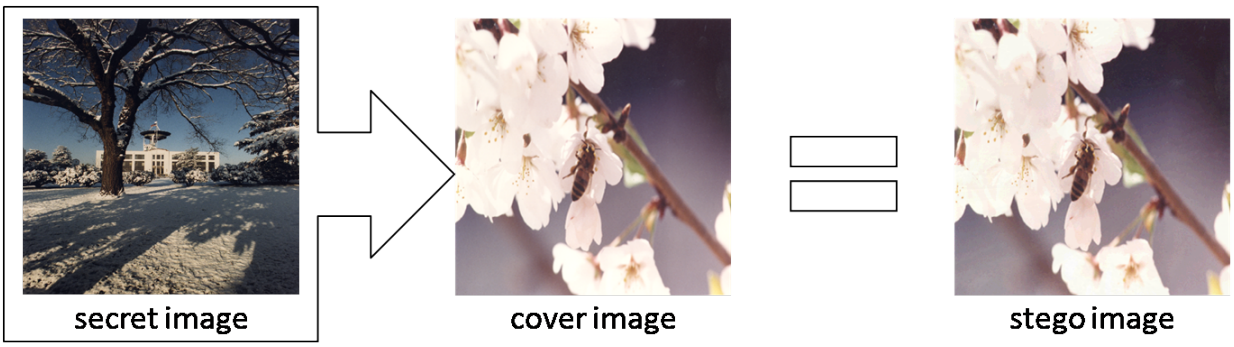
We can recover (a good approximation of) the secret pixel by taking the five least significant bits of each channel of the stego pixel and turning them into the five most significant bits of the recovered pixel. Here's the recovered pixel:



(it is common practice to set the remaining bits of the recovered pixel to 0).

Since an image is a bunch of pixels, we can exploit this technique to hide a *secret image* into a *cover image* with the same dimensions, then unsuspiciously post the resulting *stego image* to, say, Instagram, and later have a friend or accomplice recover the secret image. This is an instance of *steganography*, the art and science of hiding secret information in plain sight.

Here's an example where we are hiding the picture of an ominous satellite dish in the innocent image of a bee, pixel by pixel as we did earlier:



From it, we recover the secret image, again pixel by pixel, as we did previously:

If you look closely, you may notice that the recovered image does not look as good as the secret image. The reason for this is that, this time, we used 5 bits of (each channel of) the cover image and just 3 bits of the secret image. The number of most significant bits of the secret image to embed in the cover image is called the *quality* of the embedding: a higher quality means the recovered image will be sharper, but the stego image will not look as natural; a lower embedding quality makes it hard to tell that something is going on with the stego image, but the recovered image will not be as good (which is often fine).

Your turn! Your job is to implement the following functions:

```
pixel_t[] hide(pixel_t[] cover, pixel_t[] secret, int width, int height,
               int quality);

pixel_t[] unhide(pixel_t[] stego, int width, int height, int quality);
```

The call `hide(cover, secret, w, h, q)` returns the stego image obtained by embedding the secret image `secret` into the cover image `cover` as discussed above. Both `secret` and `cover` have size $w \times h$, and so does the returned image. The last argument, `q`, is the number of bits of the secret image to embed in the cover image; it is a number between 1 and 7 inclusive.

The call `unhide(stego, w, h, q)` returns the image recovered from the `q` least significant bits of each pixel channel in `stego`. Both `stego` and the returned array have size $w \times h$.

If the various arrays do not exactly match the given size or if the quality is invalid, your functions should abort with a precondition failure when compiled and run with the `-d` flag.

Task 3 (6 points) Create a C0 file `stego.c0` implementing the functions `hide` and `unhide`. You may include any auxiliary functions you need in the same file (a `hide_pixel` and an `unhide_pixel` may be a good idea), but you should not include a `main()` function.

You should look at `README.txt` to see how to compile and run this transformation using `stego-main.c0`. You are also strongly encouraged to write some test cases for your programs in `images-test.c0`.

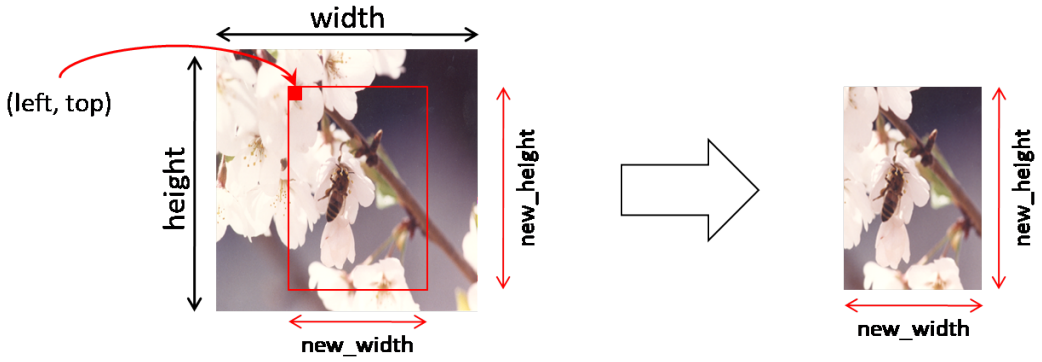
2.3 Cropping an Image

The functions `hide` and `unhide` expect the cover and the secret image to have the exact same size. But what if they do not?

In this section, we handle the situation where the width and height of the cover image are larger than (or equal to) the width and height of the secret image. We do so by writing a function to crop an image:

```
pixel_t[] crop(pixel_t[] pixels, int width, int height,
               int left, int top, int new_width, int new_height)
```

Here, `pixels` is an image of size `width` × `height`. This function returns an image of size `new_width` × `new_height` whose top-left corner is the pixel on column `left` and row `top` of the original image. Here's an example:



As in this example, the cropped image should be entirely within the input image.

There are lots of ways the input parameters can be invalid. You should write strong preconditions so that invalid inputs will abort the program when run with the `-d` flag. You should also write postconditions that allow using the returned image safely.

Task 4 (3 points) Create a C0 file `crop.c0` implementing the function `crop`. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function.

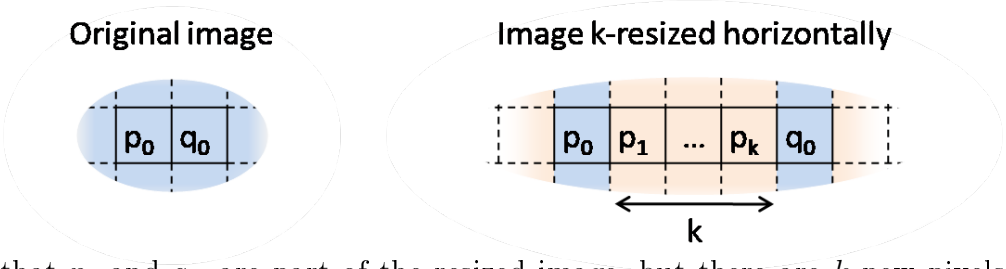
Look at `README.txt` to see how to compile and run this transformation using `crop-main.c0`. You are also strongly encouraged to write some test cases for your programs in `images-test.c0`.

2.4 Resizing an Image

Cropping helps when the cover image is bigger than the secret image, but what if it is smaller? What we will do in this case is to resize the cover image, thereby obtaining an enlarged cover image that we can then crop if needed so that it is the same size as the secret image.

Resizing an image is non-trivial, at least if we want the resized imaged to look good. In this section, we will examine an approach that, given a $w \times h$ image and an integer k, produces an image that has size about $kw \times kh$ (in fact, $((k+1)w-k) \times ((k+1)h-k)$ to be specific). We will proceed in two steps: first we will resize the image horizontally (leaving its height unchanged); then, we will resize this stretched-out image vertically, thereby obtaining our fully resized image.

Let's begin with the first step, resizing the image horizontally. The idea is very simple: we will insert k new pixels between every two pixels of the original image. Here's a view of what happens to two adjacent pixels, p_0 and q_0, of the original image:



Observe that p_0 and q_0, are part of the resized image, but there are k new pixels between them: $p_1, \ldots p_k$. The same will happen between any two adjacent pixels of the original image: on each row, k new pixels will be added after every pixel except the rightmost one (which is left alone). So, if the original image has width w, the resized image has width $w + (w-1)k = (k+1)w - k$ — the height remains unchanged.

But what should be the value of the added pixels? The idea is have them gradually transition from the value of p_0 to the value of q_0, channel by channel. So, if $k = 1$, then p_1 would be half p_0 and half q_0. If $k = 2$, then p_1 would be two thirds p_0 and one third q_0, while p_2 would be one third p_0 and two thirds q_0. In general the value of added pixel p_i is given by the following formula:

$$p_i = \frac{(k+1-i)p_0 + iq_0}{k+1}$$

This formula computes the *weighted average* of p_0 and q_0. (What does it reduce to when $i = 0$?) We perform this computation independently for each of the channels of the involved pixels — once for alpha, once for red, once for green and once for blue.

You will implement this first step of resizing an image by writing the C0 function

```
pixel_t[] stretch_horizontally(pixel_t[] pixels, int width, int height,
                               int k)
```

It returns the image obtained by inserting `k` pixels, computed as just described, between every adjacent pixels of the input image. This function should fail a precondition if called with an invalid image or a negative `k`. You will also want to have reasonable postconditions.

Once we have a horizontally resized an image by k, the second step is to do the exact same thing vertically — you can write a `stretch_vertically` function or cleverly leverage `stretch_horizontally`.

We then combine these two steps in our final resize function,

```
pixel_t[] upsize(pixel_t[] pixels, int width, int height,
                 int k)
```

which returns an image that has first been stretched horizontally by k and then stretched vertically by k.

Task 5 (9 points) Create a C0 file `resize.c0` implementing the functions `stretch_horizontally` and `upsize`. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function.

Hint: a function that transposes an image makes it very easy to write `upsize` *once you have* `stretch_horizontally` *— look it up!*

Look at `README.txt` to see how to compile and run this transformation using `resize-main.c0`. You are also strongly encouraged to write some test cases for your programs in `images-test.c0`.

At this point, you have all the pieces to hide any secret image into any cover image. If you are up for the challenge, try to write an end-to-end function that takes these two images as input and returns the resulting stego image. We will however not grade such function.

2.5 Your own image processing algorithm (Optional)

In this task, you will perform an image manipulation of your choice. The core of this transformation are three functions:

```
int result_width(int width, int height)
int result_height(int width, int height)
pixel_t[] manipulate(pixel_t[] pixels, int width, int height)
```

If `I` is the representation of an image with width `w` and height `h`, then the result of calling `manipulate(I,w,h)` should be the representation of image of width `result_width(w,h)` and height `result_height(w,h)`.

Task 6 (bonus) Create a C0 file `manipulate.c0` implementing the three functions described above: `result_width`, `result_height`, and `manipulate`. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function. You may not add arguments to `manipulate`, but you can write a separate function `my_manipulate` (or whatever) and then call your function from the `manipulate` function with some specific arguments.

You should look at `README.txt` to see how to compile and run this transformation against `manipulate-main.c0`.

If you choose to do this task, be creative! Submissions will be displayed on the Autolab scoreboard and we will highlight exemplary submissions. If you include a (small!) file `manipulate.png`, we'll run your transformation against that image; otherwise we'll run your transformation on `g5.png`.

Figure 2: Manipulate me!

A REFERENCE: the Pixel Interface

```
/*************************** Interface ****************************/
// typedef _____ pixel_t;

int get_red(pixel_t p)
/*@ensures 0 <= \result && \result < 256; @*/ ;

int get_green(pixel_t p)
/*@ensures 0 <= \result && \result < 256; @*/ ;

int get_blue(pixel_t p)
/*@ensures 0 <= \result && \result < 256; @*/ ;

int get_alpha(pixel_t p)
/*@ensures 0 <= \result && \result < 256; @*/ ;

pixel_t make_pixel(int alpha, int red, int green, int blue)
/*@requires 0 <= alpha && alpha < 256; @*/
/*@requires 0 <= red   && red   < 256; @*/
/*@requires 0 <= green && green < 256; @*/
/*@requires 0 <= blue  && blue  < 256; @*/ ;

void pixel_print(pixel_t p);
```

You can also display this interface by running the terminal command

```
% cc0 -i pixel.o0
```