15-122: Principles of Imperative Computation, Fall 2010
Assignment 1: Integers, Invariants, and Images

William Lovas (`wlovas@cs`)   Tom Cortina (`tcortina@cs`)

Out: Thursday, August 26, 2010
Due: Thursday, September 2, 2010

(Written part: before lecture,
Programming part: 11:59 pm)

1 Written: Numbers and Invariants (20 points)

The written portion of this week's homework will give you some practice working with the binary representation of integers and reasoning with invariants. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins.

1.1 Warmup

Exercise 1 (4 pts)**.** Let x be an `int` in the C_0 language. Express the following operations in C_0 using only the bitwise operators (`&`, `|`, `^`, `~`, `<<`, `>>`).

(a) Set a equal to x multiplied by 8.

(b) Set b equal to x `%` 8, assuming that x is positive.

(c) Set c equal to x with its middle 8 bits cleared.

(d) Set d equal to the intensity of the red component of x, assuming x stores the packed representation of an RGB color.

Exercise 2 (4 pts)**.** For each of the following statements, determine whether the statement is true or false in C_0. If it is false, give a counterexample to illustrate why.

(a) For every `int` x: $x + 1 > x$.

(b) For every `int` i and `int` n where $i < n$: if $n > 0$, then $i + 1 > 0$.

(c) For every `int` x and `int` y: x `%` $y < y$.

(d) For every `int` x: $x >> 1$ is equivalent to $x/2$.

Exercise 3 (4 pts)**.**

(a) How many bits are necessary to accurately represent the addition of two n-bit integers? Explain.

(b) How many bits are necessary to accurately represent the multiplication of two n-bit integers? Explain.

1.2 Reasoning with Invariants

Exercise 4 (8 pts)**.** Consider the following code:

```
int log(int n)
//@requires n >= 1;
//@ensures (1 << \result) == n;
{
    int i = 0;
    int k = n;

    while (k > 1)
    //@loop_invariant k >= 1;
    //@loop_invariant (1<<i) * k == n;
    {
        k = k / 2;
        i = i + 1;
    }

    return i;
}
```

(a) What does the expression $1 << i$ compute? Express your answer in ordinary mathematical language.

(b) Under which assumptions about i does the mathematical answer coincide with the result of the computation in the C_0 language?

(c) Explain why the postcondition (`//@ensures`) and the loop invariant are correct if we *also* require that n is a power of 2, that is, that $n = 2^m$ for some m.

(d) Write a more general postcondition and loop invariant that are still correct even if we only know that $n >= 1$ as stated above. Your loop invariant should be strong enough to ensure that your postcondition holds.

2 Programming: Image Manipulation (30 points)

For the programming portion of this week's homework, you'll write three C_0 files corresponding to three different image manipulations: `warhol.c0` (described in Section 2.2), `pixelate.c0` (described in Section 2.3), and *either* `findedges.c0` (described in Section 2.4) *or* `manipulate.c0` (described in Section 2.5), your choice.

You should submit your code electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Starter code. Download the file `hw1-starter.zip` from the course website. When you unzip it, you will find four C_0 files—`warhol-main.c0`, `pixelate-main.c0`, `findedges-main.c0`, and `manipulate-main.c0`—corresponding to the four programming problems below.[1] Each file has a `main()` function that will read an image from disk, call your code on its representation, and then write the result image back to disk. You need not submit these files when you hand in your code, and the files you submit should not include `main()` functions.

In addition, you will find a sample manipulation `remove-red.c0`, which removes the red channel from each pixel of an image, and its associated main file `remove-red-main.c0`. This sample provides a complete program that you can compile and execute, and you may pattern your code after the code in `remove-red.c0` if you find it convenient to do so. (The code for the `remove_red` function also appears in Appendix A.)

Finally, you will also see an `images/` directory with some sample input images and some sample outputs for some of the manipulations. On a Linux cluster machine, there are several programs you can use to view the images, including `display`, `gpicview`, `qiv`, `eog`, and `gthumb`. Play around and find one you like.

Compiling and running. For this homework, we are providing you with a script called `cc0hw1` to compile your code because there are many required library options. The script is located in the same directory as the `cc0` command and accepts the same options as `cc0`, so once you have `cc0` working, `cc0hw1` should work the same. For example, recall that we can enable dynamic annotation checking in the compiled binary by running `cc0 -d`. Similarly, we can enable dynamic checking when compiling homework 1 files by running `cc0hw1 -d`.

To compile one of your files against one of our `main()` functions, just specify both files on the command line. Let `<file>` be one of `warhol`, `pixelate`, `findedges`, or `manipulate`; you can compile your `<file>.c0` on any Andrew system by running the command

```
cc0hw1 <file>.c0 <file>-main.c0 -o <file>
```

[1] The `manipulate` task has a slightly more complicated spec, so there is additionally a `manipulate-starter.c0` file to get you started if you choose to do that task. See Section 2.5 for details.

from the directory created by the starter code zip file. This will will place the compiled binary in the file `<file>` rather than the usual default `a.out`. Once you've compiled `<file>` in this way, you can run it with the command

```
./<file>
```

The file so produced will expect some options of its own, at the very least an option `-i <input file>` specifying the input image to manipulate. If you run one of the programs without any arguments, you will get a short usage message explaining the options particular to that program.

As a concrete example, you can compile the `remove-red` filter with dynamic checking and run it on the sample image `g5.jpg` in the `images/` directory by running the following commands in sequence:

```
cc0hw1 -d remove-red.c0 remove-red-main.c0 -o remove-red

./remove-red -i images/g5.jpg -o images/g5nored.jpg
```

If you have any problems compiling or running your code as described here, you should contact the course staff.

Submitting. Once you've completed some files, you can submit them by running the command

```
handin -a hw1 <file1>.c0 ... <fileN>.c0
```

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include `//@requires`, `//@ensures`, and `//@loop_invariant` annotations in your program. You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct.

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

2.1 Image Manipulation Overview

The three short programming problems you have for this assignment deal with manipulating images. An image will be stored in a one-dimensional array of integers, where each integer is a 32-bit value representing one pixel of the image. 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":

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

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

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

Each pixel in the array is a 32-bit integer that can be broken up into 4 components with 8 bits each:

$$a_1a_2a_3a_4a_5a_6a_7a_8\ r_1r_2r_3r_4r_5r_6r_7r_8\ g_1g_2g_3g_4g_5g_6g_7g_8\ b_1b_2b_3b_4b_5b_6b_7b_8$$

where:

$a_1a_2a_3a_4a_5a_6a_7a_8$	represents the alpha value (how opaque the pixel is)
$r_1r_2r_3r_4r_5r_6r_7r_8$	represents the intensity of the red component of the pixel
$g_1g_2g_3g_4g_5g_6g_7g_8$	represents the intensity of the green component of the pixel
$b_1b_2b_3b_4b_5b_6b_7b_8$	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).

For example, a pixel that is completely opaque with only green at its maximum intensity would be stored as the integer 0xFF00FF00. An opaque pixel that is medium gray would be 0xFF7F7F7F (equal parts red, green, and blue at medium intensity).

For the rest of the assignment, we will work under the assumption of a type definition that makes `pixel` an alias for `int`:

```
typedef int pixel;
```

Since `int`s are used for many other things (like the width and height of an image, for example), a type alias is useful for distinguishing those instances where we mean to interpret an `int` as an RGB pixel. You should include this `typedef` in your code and use the `pixel` type when appropriate.

2.2 Required: Warholization

In this problem, you will create an image effect similar to the screenprintings of Andy Warhol, a famous "pop artist" who grew up in Pittsburgh and studied commercial art right here at Carnegie Mellon[2]. Warhol was particularly well-known for painting extremely common everyday icons like soup cans or famous celebrities. His silkscreens of celebrities often involved reduced color spaces and serially repeated images.

Your task here is to implement a function that takes as input an image of size $w \times h$ and create a "Warhol" image of size $2w \times 2h$ that contains the same image repeated four times, the top left image containing only the red component of the original, the top right containing only the green component, the bottom left containing only the blue component, and the bottom right containing a gray-scale rendition of the original image. A sample image is shown in Figure 1 before and after "Warholization".

Recall that you can extract the various components of a pixel's color using bitwise shifts and masks. To compute the gray-scale value for a pixel in the new image, average the red, green, and blue components of the original pixel and set the red, green, and blue components of the new pixel to this average value, leaving the alpha component unchanged.

Task 1 (10 pts)**.** Create a C_0 file `warhol.c0` implementing a function `warhol` matching the following prototype:

```
pixel[] warhol(pixel[] pixels, int width, int height);
```

where `width` and `height` represent the width and height of the original input image.

The result array should be the array representation of the "Warholized" image. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function.

2.3 Required: Pixelation

In this problem, you will implement a pixelation routine that achieves an effect similar to the "anonymizing" blur sometimes seen on television news and documentary productions. You can see an example in Figure 2.

Given an ordinary image of size $w \times h$, you can create a pixelated image of size $w \times h$ using a block size of b by replacing each pixel in a $b \times b$ square with the average color value of every pixel in that square. Starting from the $b \times b$ square in the top left of the original image, average all of the red components of each pixel together and store the average value as the red component of each pixel in the same $b \times b$ square of the result image. Repeat this same procedure for the blue and the green components. Leave the alpha component unchanged.

Then repeat this process for the $b \times b$ square that is completely to the right of the top left one, and so on in turn for each non-overlapping $b \times b$ square.

[2]Then known as the Carnegie Institute of Technology

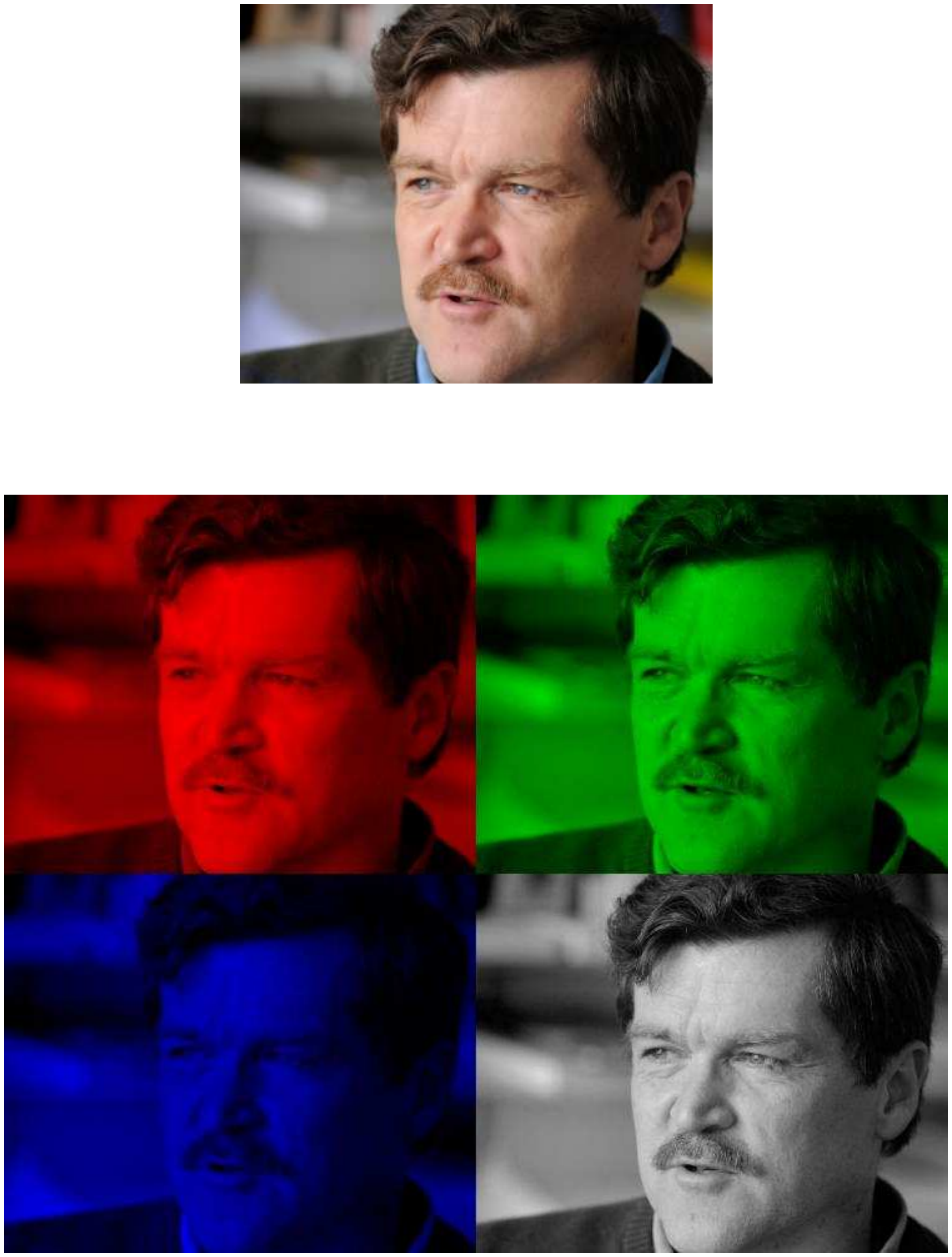

Figure 1: A famous computer scientist and his rendition by our Warhol filter.

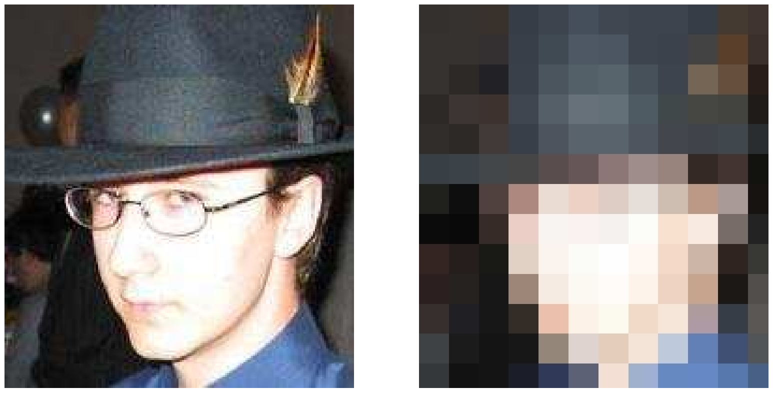

Figure 2: A shady character, before and after anonymization with a block size of 16.

Note that as you reach the right side or bottom of the image, you might not have enough pixels to form a complete $b \times b$ square; if this occurs, average as many pixels as you can based on where the $b \times b$ square would have been if the image were larger.

Task 2 (10 pts)**.** Create a C_0 file `pixelate.c0` with a function `pixelate` matching the following prototype:

```
pixel[] pixelate(pixel[] pixels, int width, int height,
                 int blocksize);
```

This function should implement the pixelation algorithm described above, given an array `pixels` representing an image of width `width` and height `height`, using pixelation blocks of size `blocksize` × `blocksize`.

The result array should be the representation of the pixelated image. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function.

2.4 Alternative 1: Edge Detection

Note: You should submit code *either* for Task 3 in this section *or* Task 3 below (in Section 2.5), *not both*. You will only receive credit for one of the two tasks.

In this problem, you will write a program to detect *edges*, which are the curves in an image that separate areas of high contrast. The result of edge detection is an image where each pixel is either completely black (0xFF000000) or completely white (0xFFFFFFFF), depending on whether it occurs as part of an edge or not. Figure 3 shows a sample image and its edges.

To perform edge detection on an image of size $w \times h$, create a result image of the same dimensions and consider each pixel of the original image in turn. For each one, compute the difference in brightness between the pixel above and the pixel below, and compute the difference in brightness between the pixel to the left and the pixel to

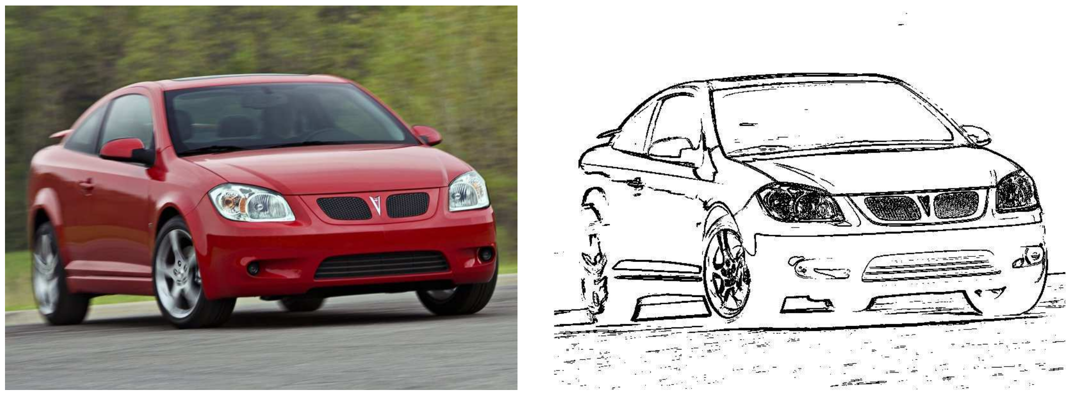

Figure 3: Tom's Pontiac G5, a digital car for the digital world, and its edge detection with a threshold of 500. (Not pictured: Tom.)

the right. (The brightness of a pixel is the same as its gray-scale value: the average of the three color components.) Square each of these differences and add them together. If the sum exceeds the given threshold value, then the pixel is on an edge and the corresponding pixel in the result image should be black; otherwise, the pixel is not on an edge, and its corresponding pixel in the result should be white.

Essentially, what this algorithm amounts to is looking for significant changes in brightness in either the vertical or horizontal direction. If there is a significant enough change, the pixel is determined to be on an edge.

Note that the algorithm breaks down for pixels along the border of the original algorithm. (Think about why!) For these pixels, you may simply set the corresponding pixels in the result image to white.

Task 3 (Alternative 1, 10 pts)**.** Create a C_0 file `findedges.c0` with a function `findedges` matching the following prototype:

```
pixel[] findedges(pixel[] pixels, int width, int height,
                  int threshold);
```

This function should implement the edge detection algorithm described above, given an array `pixels` representing an image of width `width` and height `height`, using a threshold value of `threshold` to decide when a pixel is on an edge.

The result array should be the representation of the pixelated image. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function.

2.5 Alternative 2: Your Choice

Note: You should submit code *either* for Task 3 above (in Section 2.4) *or* Task 3 below, *not both*. You will only receive credit for one of the two tasks.

Task 3 (Alternative 2, 10 pts). Write a function `manipulate` that performs an image manipulation of your choice matching the following prototype:

```
pixel[] manipulate(pixel[] pixels, int width, int height);
```

You will also have to write two small functions that express the width and height of the result of your manipulation in terms of the width and height of the input image:

```
int result_width(int width, int height);
int result_height(int width, int height);
```

The starter code archive contains a file `manipulate-starter.c0` with empty stubs for these functions and a main file `manipulate-main.c0` that you can compile against to get a binary that runs your manipulation.

If you choose this task, be creative! A "judges' prize" will be awarded to the student whose submission "impresses" the course staff the most. (Of course, we reserve the right to decide for ourselves what that means!)

A Sample Code: Remove Red Channel from an Image

```
/* make pixel a type alias for int */
typedef int pixel;

pixel[] remove_red (pixel[] A, int width, int height)
//@requires \length(A) >= width*height;
//@ensures \length(\result) == width*height;
{
  int i;
  int j;
  pixel[] B = alloc_array(pixel, width*height);

  for (j = 0; j < height; j++)
  //@loop_invariant 0 <= j && j <= height;
  {
      for (i = 0; i < width; i++)
      //@loop_invariant 0 <= i && i <= width;
      {
        // Clear the bits corresponding to the red component
        B[j*width+i] = A[j*width+i] & 0xFF00FFFF;
      }
  }
  return B;
}
```