Colorado Mesa University
Computer Science and Engineering

CSCI112 Data Structures

Practicum Week 4

Due Thursday Practicum of your Week 4.

Background:

In this practicum you will gain some more experience implementing classes and pointers.

Preamble:

In class we talked about binary files. For the most part binary files are a nice way to store fixed size records and perform random I/O. Binary files have more diverse uses then in database programs.

Consider the Portable Pixmap image format (PPM). The PPM image format stores an image as a collection of pixels along with some header information.

Each pixel has three components to compute its colour – these being red, green and blue. The PPM format requires each pixel component (R,G,B) to be 1 byte long – therefore making an individual pixel 3 bytes in total.

If you look inside a PPM file e.g. `iphone.ppm` (provided) with a text editor you will see the following:

```
P6
#.
284 428
255
fffUUUDD333
```

The first four lines represent header information. Of these four lines the `284` and `428` represent the horizontal and vertical lengths of the image. The `255` represent the bit depth i.e. the maximum colour value. The data following the `255` represents pixel information.

From this information we can learn a lot about an image. We now understand that there are `284` x `428` pixels in this image (totalling `121552`). How do we represent this image inside the computer?

Well first you need to represent a single pixel. Each pixel has three 1 byte components for R, G and B component values. Given this information we can use the following to represent a single pixel (remember a char is 1 byte wide).

```
struct pixel
{
    unsigned char r;
    unsigned char g;
    unsigned char b;
};
```

Now that we can represent a single pixel – how can we represent an image which has a resolution of 284 x 428. Well this can be best though of as a two dimensional array of type pixel.

```
pixel picmap[428][284];
```

Each pixel can be manipulated independently. So for example `picmap[100][2]=` modifies the third pixel (column) on the one hundred and one'th row.

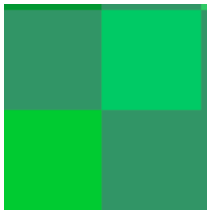
If you read the source code provided in `enlarge.cpp`, the code to read images up to a size of `1500` x `1500` into an array of pixels has been provided. The code additionally reads and interprets the header information contained in the image.

After reading the header information, the code reads each pixel and puts it into the corresponding array element till done. It should be noted that each pixel is represented in order in the file – this means the first 3 bytes of the pixel data represent the first pixel, the next 3 bytes represent the second pixel and so on.

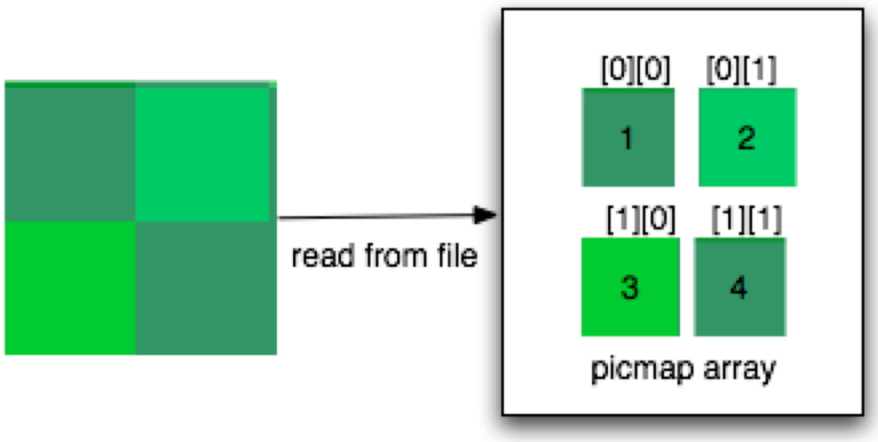
The code that reads the pixels from the file into the `picmap` array is:

```
    for (int i = 0; i < ydim; i++)
    {
        for (int y = 0; y < xdim; y++)
        {
            ins.read((char*)&picmap[i][y], sizeof(pixel));
        }
    }
```

So how does this code work. Lets consider a PPM image that looks like this:



Each square represents a single pixel. In this case the resolution of the image is 2 x 2. Given the code above each pixel element is read into the appropriate `picmap` array element. Remember when reading from the binary file, each pixel is 3 bytes long – so we read `sizeof(pixel)`'s. We repeat this process until there are no more pixels to read i.e. we have done the entire resolution. The illustration below shows where each pixel goes in the array.



So that's pretty easy…

The program I provide works as follows. The source code provided takes the name of an image to read in. The image that is read in is stored in raw form in the `picmap` buffer.

Once the image is read in, the user is prompted for a scaling value. You can enlarge the picture, 1, 2, 3, or 4 times. If you enlarge a picture that is `100` x `100` pixels x 3 times you get a picture that is `300` x `300`.

The next thing you are prompted for is where to write the output. So after the image is enlarged it is written to the file nominated here.

It contains the code that enlarges and writes the image to the destination file.

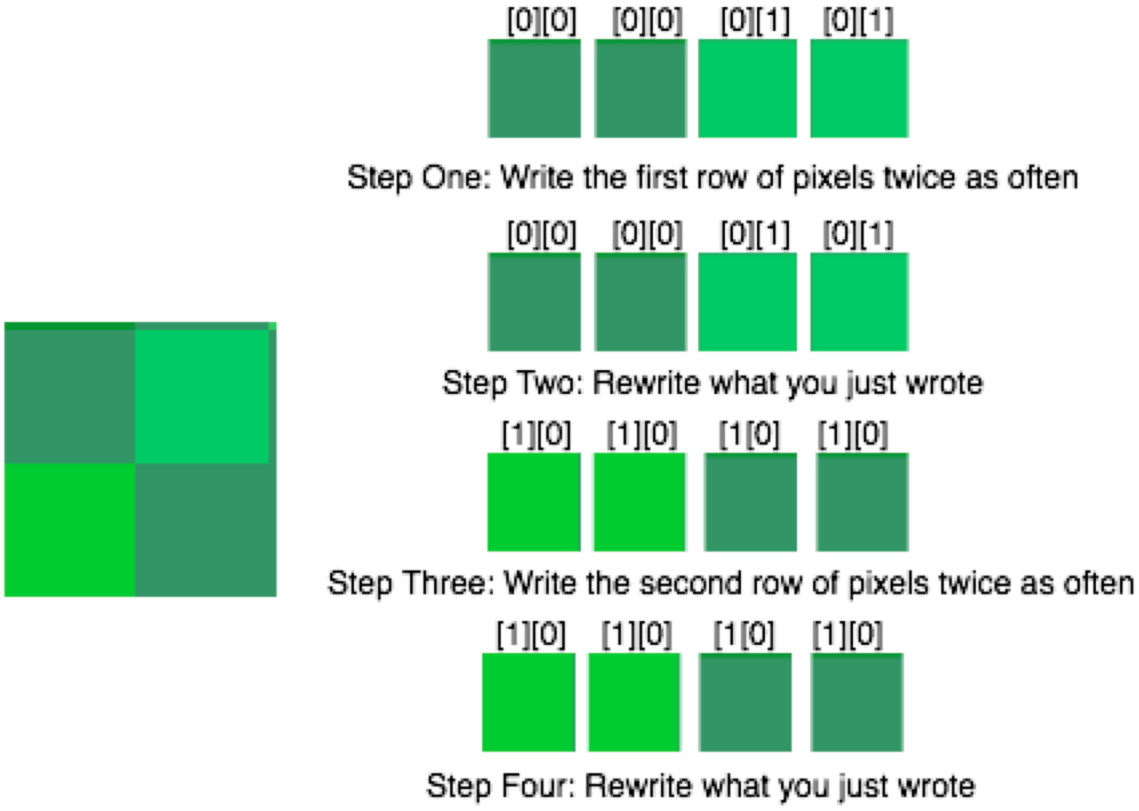
The enlarging and writing of the image is to take place in the function

```
bool writefile(char filename[], int xdim, int ydim, int scale,
                                                pixel picmap[XMAX][YMAX])
```

The function takes the filename of the file you are writing too, along with the integer horizontal and vertical dimensions of the image. It also takes a scaling value that is how many times you wish to enlarge the image. Finally the function takes the array representing the image.

The code provided opens the file and writes the appropriate header information..

How does it do this? Consider our example from above with the image that is 2 x 2 pixels. Lets say we are scaling it two times.



You simply write the pixels one after the other to the file. The array indices refer to locations in the picmap array. Remember when writing each pixel you are writing 3 bytes.

The `openfile` function to give you more information on the pixels and their arrangement in memory.

Practicum Task

In the preamble above you see the description of a program which read in PPM image files and scale them accordingly before writing them back out to disk.

In this week.s practicum your job is to modify this code. The task consists of a number of small steps.

Step One

As you may recall, a PPM image consists of a header followed by a series of pixels. The number of pixels in the image is documented in the image header as the length and width values for the image. As is the case in `enlarge.cpp` the pixels are read into a buffer of fixed size, then the appropriate manipulation is performed in this case scaling.

The problem however with `enlage.cpp` is simple… It can not handle images greater then 1000 x 1000 pixels. The solution to this problem is to allocate this buffer dynamically at runtime.

Make appropriate modifications to `enlarge.cpp` so that an appropriate size buffer is allocated on the heap at runtime for the image being read in – naturally this is based on the size of the image being read in. In the event that you cannot allocate the memory on the heap your program should terminate.

In addition to this, make suitable modifications to the program so that the image can be scaled more then 4 times. Pick a sensible number here for the scaling limit – maybe 10. How big a scale factor do you need before the program will not allocate the segment on the heap?

Finally when your program is finished, be sure to deallocate all memory you allocated for the program.

Please note, you are allowed to change as much code as you like in `enlarge.cpp`. That said you are not allowed to move the `picmap` buffer used to store your image at runtime into global space.

Once complete, submit your modified `enlarge.cpp`.

Step Two

In reality code is not written like this – more often then not a developer does not know the internals of a library – it is best to consider these routines that manipulate the image as library routines. Your next job is to rewrite `enlarge.cpp` using object based design approaches discussed in class. Using the object based design approach two objects stand out in this problem. One is an image, the other is a pixel. Remember a single image consists of a number of pixels. Both the image and pixel are perfect candidates for classes.

However for the purpose of simplicity, in this task we are only going to use one class representing an image. A pixel will be implemented using a `struct` (it could just as easily be implemented as class).

A image is represented using the class definition:

```
class image
{
     public:
          image();
          ~image();
          void openfile(char filename[]);
          void scaleimage(int factor);
          void writeimage(char filename[]);
     private:
          struct pixel{
               unsigned char r;
               unsigned char g;
               unsigned char b;
          };

          pixel* picmap;
          int xdim, ydim;
          bool status;
};
```

The class is very straight forward. The constructor/ destructor are responsible for initializing/ destroying private members. The `openfile` method takes a string representing a PPM file, attempts to open it and read the contents of the PPM bitmap into a dynamically allocated buffer associated with the `picmap` pointer. **PLEASE NOTE PICMAP WILL BE A SINGLE DIMENSION ARRAY WHEN ALLOCATED – YOU WILL HAVE TO DO ARITHMETIC TO REPRESENT THE 2D ARRAY IN THIS.** If for any reason the operation fails, the program should terminate. It should be noted that `loadfile` can only be invoked once on any instance of class `image` – you should enforce this (look at the private members for a tip on how to do this).

The next function is the `scaleimage` function, which allocates a buffer on the heap, *factor* x the number of pixels in the image. Once this is done the already loaded `picmap` is scaled and stored in this newly assigned heap segment. At the end of this process, `picmap` should be a pointer to the new buffer which represents the scaled image. If for any reason the operation fails – terminate the program. Note you can only scale an image after it has been loaded. The private member status can help you decide this if used correctly.

The final function which has to be implemented is `writeimage`. The function takes a filename and writes the contents of `picmap` – regardless of whether or not it has been scaled to a file.

Implement this class accordingly. Complete the `main.cpp` function provided (without changing any of the pre-existing code). On completion main.cpp should mimic the behavior of `enlarge.cpp`. Be sure to implement appropriate error checking.

`enlarge.cpp`
`image.cpp`
`image.h`
`main.cpp`

Submission

At the end of the class show your work to your tutor. Your tutor may ask you to demonstrate various aspects of your mark and/or explain things.