Project 2

In this project you will take what you have learned about pointers and memory manipulation and use those skills to do some image processing.

Project parameters

Background

A *bitmap*, one of the main ways of representing pictures, is a two-dimensional grid of pixels, each having its own color. Most image formats you are likely familiar with (e.g. JPEG, PNG, GIF) store their data in the form of a bitmap.

How colors (pixels) are represented can vary, but a common method ("24-bit color") is to store a pixel in three bytes with one byte each for blue, green, and red, in that order: `BB GG RR`. On the other hand, as we have already seen in class, inside a program we normally represent a color using the bottom three bytes of an `int` value: `0xRRGGBB`.

Here are some examples of colors in both forms:

	Color	Pixel (as saved in file)	Pixel (as an `int`)	Type
☐	Red	`00 00 FF`	`0xFF0000`	
☐	Green	`00 FF 00`	`0x00FF00`	Primary colors
☐	Blue	`FF 00 00`	`0x0000FF`	
☐	Yellow	`00 FF FF`	`0xFFFF00`	
☐	Magenta	`FF 00 FF`	`0xFF00FF`	Secondary colors
☐	Cyan	`FF FF 00`	`0x00FFFF`	
☐	Black	`00 00 00`	`0x000000`	
☐	Medium gray	`80 80 80`	`0x808080`	Grayscale (`BB = GG = RR`)
☐	White	`FF FF FF`	`0xFFFFFF`	
☐	Night	`51 15 21`	`0x211551`	
☐	Copper	`1F 58 9D`	`0x9D581F`	
☐	Snow (a.k.a. "white")	`FF FF FF`	`0xFFFFFF`	
☐	Flint	`20 18 10`	`0x101820`	Westminster brand colors
☐	Birch	`DE F1 F1`	`0xF1F1DE`	
☐	Thistle	`C7 52 82`	`0x8252C7`	
☐	Sky	`E2 B5 00`	`0x00B5E2`	

In this project, you will write code that reads a bitmap image from a .bmp (Windows Bitmap) file. Although this file format is less common than it used to be, it is very easy to work with since (unlike JPEG, PNG, and GIF) the image data is normally uncompressed. Having read in the image, your program will transform the image data in a few different ways, then save it back to .bmp files.

Learning goals

By doing this project, you will gain experience with the following:

Binary file I/O
Reading and writing binary files (in this case .bmp files)

Memory-mapped I/O
Mapping the contents of a file into memory so that its bytes can be accessed directly through pointers

Pointer manipulation
Using pointers to access specific parts of the .bmp file and pull out (or insert) data

Dynamic memory allocation
Allocating memory dynamically using `malloc()` to hold image pixel data

Roadmap

The purpose of this project is to map a .bmp file into memory, read its contents into a dynamically-allocated array of pixels, transform that image data in a few different ways, and finally save it to some output files. To do this you will do the following steps.

Step 1
Write the code to map an input file into memory so that its contents can be accessed through pointers.

Step 2
Following the .bmp file format, read image data (such as its width and height) and then load the image's pixel data.

Step 3
Write the code to do the inverse of Steps 1 and 2: map an output file into memory and then write image data to it.

Step 4
Write functions that transform bitmap images without changing their dimensions (that is, just altering the colors).

Step 5
Write functions that transform bitmap images in ways that alter the image dimensions (e.g. stretching).

Extra credit

If you want a few extra points, clean up the remaining memory leaks and add some more bitmap operations!

If you are not able to complete a step

This project is structured such that, except for the final step, you can complete each of the steps individually. If you get stuck on Step 1, then ask questions, but while waiting for assistance you can work on Step 2. In fact, if you wish you could do Steps 1–5 in any order, although the order they are in builds up to the full program naturally.

Grading

Grading of this project is described in the Canvas assignment for this project.

Step 1

Goal Write the code to map an input file into memory so that its contents can be accessed through pointers.

Functions needed for this step

```
int open(char *filename, int flags)
```

Opens a file for reading or writing. The parameters are

- `filename`: the name of the file to open.
- `flags`: how you intend to use the file. For reading from a file, use `O_RDONLY` (“read-only”).

The function returns an integer “file descriptor” (`fd`) — this is used in all subsequent I/O-related calls to refer to this opened file. If an error occurred, -1 is returned.

```
int close(int fd)
```

Closes a file previously opened with `open()`. If an error occurred, -1 is returned; otherwise it returns 0.

```
int fstat(int fd, struct stat *statbuf)
```

Returns information about a file previously opened with `open()`. The parameters are

- `fd`: the file descriptor of the file to get information about
- `statbuf`: pointer to a `struct stat` that will be filled with information about the file.

The `stat` structure has a number of fields that can tell you the owner of a file, its permissions, and so on. (If you are interested, type “`man 2 stat`” in a command line on the server.) For our purposes, the important field is `st_size`, which is the size of the file (in bytes).

This function returns -1 if an error occurred and 0 otherwise.

```
void *mmap(void *addr, long length, int prot, int flags, int fd, long offset)
```

Maps a previously opened file into memory, so that you can treat the contents of the file as any other pointer. The parameters are

- `addr`: a hint about where in memory you would like the file to be mapped. (Should almost always be `NULL`.)
- `length`: how many bytes of the file should be mapped into memory. (Typically this would be the file size.)
- `prot`: the protection mode of the mapped memory. For reading from a file, use `PROT_READ`.
- `flags`: what relationship the mapped memory should have to the original file. Normally `MAP_SHARED` is the correct choice, meaning that whatever is in the file will be reflected in memory (and vice-versa).
- `fd`: the file descriptor of the file to map into memory.
- `offset`: the offset within the file of the mapped memory. (Normally this will be 0.)

Assuming the call to `mmap()` succeeds, then afterwards you can access the contents of the file through the pointer that it returns. If it fails, then it returns `MAP_FAILED`.

```
void perror(char *s)
```

Prints a user-readable version of the last error that occurred when calling a system function. The string parameter `s` is printed before the error as a means of saying which component of the program caused the error; normally we just use `NULL` to print just the error message itself.

Since C doesn't have exceptions, it is important to check the return value after every function that could fail (which includes all of those above!). If the return value indicates an error, then use `perror()` to print an error message so you have some idea of what the problem was.

Directions

Make a directory to hold your Project 2 files and copy `project2.c` from `/var/project02` into it. Also copy `Makefile` from the same directory; this has predefined rules for compiling your project and running it through the testers I've provided. For this step, you'll be filling in the code for the `map_file_for_reading()` function.

The purpose of this function is to open a file and map it into the program's memory so that its contents can be accessed through a pointer. The function takes a filename as a parameter and returns a void pointer to the mapped memory. If any errors happen along the way (for instance, trying to open a file that doesn't exist or that the user doesn't have permission to open), the function should print an error message and return `NULL`.

The steps needed for mapping a file into memory for reading are the following:

A. Use `open()` to open the file for reading. The result is a file descriptor (`fd`).

B. Use `fstat()` to determine the size of the file. (This is needed so that we can later tell `mmap()` how many bytes of memory should be mapped.)

C. Call `mmap()` to map the file into memory. The function has a bewildering collection of parameters, but the information above tells you what most of them should be!

D. Close the file using `close()`. (Once the file is mapped into memory, it is safe to close.)

E. Return the pointer returned from `mmap()`.

Make sure you check the return value of every function (except `close()`) for errors. Since C does not have exceptions, it is the responsibility of the programmer to write the code to check these things! If an error does occur, your function should first use `perror(NULL)` to print an error message, then return `NULL`.

Testing your code

When you are ready to test your code, run it through the Step 1 tester using the following command:

```
make test1
```

This will compile your code, and if that succeeds, will link it with the Step 1 tester and run the result. The tester will verify that your `map_file_for_reading()` function works correctly in the case that it should work; it does a little testing of a case where it should *not* work (mapping a file that the program doesn't have permission to read), but there's only so much it can do to simulate errors, so **it is your responsibility to make sure that your function checks all the return values and behaves appropriately**.

There are a few other `make` commands you can use as well. `make compile` will compile your project (this is just a shortcut for the usual `gcc` command); `make run` will compile your program (if it needs to be compiled) and then run it, similar to the "Run" button in an IDE. Of course you can also use the usual `gcc` commands that you know to do all this manually.

Step 2

Goal Following the .bmp file format, read image data (such as its width and height) and then load the image's pixel data.

Now you will write the code to import the image data, now that we can map it into memory.

Note: this step is by lengths the longest (and hardest) step of the project. There are a lot of moving pieces involved in reading a structured binary file like a .bmp file! However, this step is not worth any points more than any of the others, so if you are struggling, you might want to work on some of the other steps. As always, please ask questions on Slack when you get stuck!

Write helper functions

Begin by filling in the code for the `rgb_to_pixel()` function, which takes red, green, and blue components (each a single byte, 0–255) and places them together in an integer `p` in the usual order: `0xRRGGBB`. Also fill in `pixel_to_rgb()`, which does the reverse. (We have written these functions in class a couple different ways now, so feel free to borrow from your notes.)

Once you have these functions written, compile and run your code against the Step 2 tester:

```
make test2
```

It will complain about `read_bitmap()` not working because you haven't written it yet, but you can disregard that error. For now just make sure that your `rgb_to_pixel()` and `pixel_to_rgb()` pass the tests.

Checking the magic

Every .bmp file is supposed to start with two bytes "BM" (that is, `42 4D` in hexadecimal). This is called "magic," and is a quick way to check whether the file is actually a bitmap. The first job of your `read_bitmap()` function is to check that the file does start with "BM", and signal an error (by returning -1) otherwise.

The `read_bitmap()` given to you starts by casting the `bmp_file` parameter to a `byte *` called `file`, allowing us to access the file byte by byte. (Near the beginning of the file there's a `typedef` that makes `byte` a synonym for `unsigned char` — this is helpful because on the server a `char` is signed by default and we don't want negative values!)

After the cast, fill in some code that checks whether the first byte is `'B'` and the second is `'M'`. That is, treat `file` as an array (of bytes) and look at its first two elements. (You can write `'B'` and `'M'` directly in your program for the values to check against.) If the first two bytes are "BM", then return 0, and otherwise print an error message and return `-1`.

After your code is written, run it against the step 2 tester again; it should now pass the `Attempting to read a file that isn't a bitmap` test (but it will still fail on the later tests).

Reading the image parameters

The next step is to read the essential data about the image, primarily its width and height. There are a number of different formats for a .bmp file, but the most common (and the one we'll use here) looks like this:

	Offset	Bytes	Field name	Notes
➡	0	2	Magic	Should always be "BM"
	2	4	File size	The total size of the file in bytes
	6	4	Reserved	Safe to ignore
➡	10	4	Pixel data offset	The location in the file where the pixel data begins
	14	4	Header size	Acts as a version number for the .bmp file
➡	18	4	Bitmap width	Width of bitmap in pixels
➡	22	4	Bitmap height	Height of bitmap in pixels
	26	2	Color planes	Should always be 1 (safe to ignore)

	Offset	Bytes	Field name	Notes
➡	28	2	Color depth	Color depth (bits per pixel). For us, should always be 24.
➡	30	4	Compression method	We will not deal with compressed bitmaps, so should always be 0
	34	4	Image size	Size (bytes) of raw image data (safe to ignore)
	38	4	Horizontal resolution	Resolution in pixels per meter (safe to ignore)
	42	4	Vertical resolution	
	46	4	Palette size	Number of colors in palette (safe to ignore)
	50	4	Number of important colors	How many colors are "important" (ignore)
➡	*(variable)*	*(variable)*	Pixel data	For the files we work with, a sequence of `BB GG RR` values

The fields marked with an arrow are important ones that you should use in your function as indicated below. Others can be ignored for our purposes.

- *Magic:* signal an error if it's not `"BM"` (you already did this above).
- *Pixel data offset:* grab this value and save it in a variable; you'll need it later. (See the note below on how to do this.)
- *Bitmap width and height:* save these in the `struct bitmap` that `bmp` points to.
- *Color depth:* make sure this is 24; if it isn't, print an error message and return -1.
- *Compression method:* make sure this is 0; if it isn't, print an error message and return -1.
- *Pixel data:* we'll handle this in the next section.

Reading fields from the file

In order to read any of these fields (for example, the bitmap width) you need to know two things: its offset (where it is in the file) and its size (which will determine what data type to use for it). The bitmap width has a size of 4 bytes, so you will want to use an integer type whose size is 4 bytes; `int` fits the bill.

However, our pointer is a `byte *`. The easiest way to get the pixel data offset as an `int` value is to do the following:

1. add the needed offset (18 for bitmap width) to the pointer (`file`) to move to the right address;
2. cast the result to a pointer of the appropriate type (`int *` for bitmap width since it's 4 bytes) so that we are looking at the bytes as a whole rather than individually; and finally
3. use `*` to dereference, yielding a value of whatever type we decided on.

You will need to use several pairs of `( )` in your code to make sure those three steps happen in the right order!

Add code to your `read_bitmap()` function to extract the width and height and place them in the corresponding fields of the `struct bitmap` pointer parameter `bmp`. Compile and run against the tester again; if it complains that `read_bitmap() did not initialize pixels!` but does not give any errors about the width or height, then you've reading the width and height correctly.

After you have the width and height working correctly, handle the remaining important fields as the bulleted list above indicates. (Keep in mind that the color depth is a two-byte quantity, not four, so you will need to use a data type other than `int` to access it!)

Reading the pixel data

Once you are able to read all the fields correctly, it's time to read the actual pixels of the image. The pixel orders section describes how pixel data is laid out in the .bmp file, as well as how it should be stored in the `struct bitmap`'s `pixels`.

Write the code

Write code that reads pixels from the `file` pointer and places them into the `pixels` field of the `struct bitmap` pointer parameter `bmp`:

- First you will need to use `malloc()` to allocate memory for the pixel data (the `pixels` field of `bmp`). Note that we are *not* doing anything weird with the stride here (that's just in the file) so the memory needed is simply width × height × `sizeof(int)`.

- The easiest way to copy the pixel data over is using nested `for` loops. I recommend something like the following for an overall structure.

```
// Compute the stride (number of bytes in one row)
int stride = (24 * width + 31) / 32 * 4;

for (int y = 0; y < height; ++y)
{
    // Calculate where this row of pixels begins in the file
    // and assign it to the "row" variable
    // (Use the pixel data offset and stride)
    // Remember that the file stores rows from bottom to top!
    byte *row = ...

    for (int x = 0; x < width; ++x)
    {
        // Locate the B, G, and R bytes for the pixel (x, y)
        // (use the row variable - each pixel will be three bytes)

        // Assemble the R, G, B into a pixel value by using rgb_to_pixel()

        // Place the pixel (x, y) at the appropriate place in the pixels arr

    }
}
```

- Don't forget that the file is storing rows of the image from bottom to top, but we want to do the reverse in the pixels array!

Once your code is ready to test, run it against the step 2 tester once again. It will test your code against three images (each of which is available for you to look at on your own to help chase down bugs).

Common errors

If you are having issues with this part, please check whether it matches any of the following. This section will continue to be updated as students ask the instructor for assistance.

→ **Tester fails with "pixel incorrect" message — and every row of pixels seems to be the same**

Make sure you have calculated the stride correctly. In particular, it should be

```
stride = (24 * width + 31) / 32 * 4
```

not

```
stride = (24 * width + 31) / (32 * 4)
```

Step 3

Goal Write the code to do the inverse of Steps 1 and 2: map an output file into memory and then write image data to it.

In this step you will fill in the `map_file_for_writing()` and `write_bitmap()` functions, which together will allow us to save bitmap files.

Functions needed for this step

```
int open(char *filename, int flags, int mode)
```

This is the same function we used in Step 1, but when we're using it for writing a file, we add the third parameter (`mode`). The parameters are

- `filename`: the name of the file to open.
- `flags`: how you intend to use the file. For writing to a file, use `O_RDWR | O_CREAT` ("read/write, create if necessary").
- `mode`: what permissions you would like the file to have if it will be created (these are the same as what we use in the `chmod` command). Typically you would use 0644 or 0600.

The function returns an integer "file descriptor" (`fd`) — this is used in all subsequent I/O-related calls to refer to this opened file. If an error occurred, -1 is returned.

```
int close(int fd)
```

Closes a file previously opened with `open()`. If an error occurred, -1 is returned; otherwise it returns 0.

`int ftruncate(int fd, int bytes)`

Sets the size of a file (previously opened with `open()`) to the given number of bytes. (If the file was previously larger than `bytes`, it is truncated; if it was shorter, then the remaining content is filled with zeros.

This function returns -1 if an error occurred and 0 otherwise.

`void *mmap(void *addr, long length, int prot, int flags, int fd, long offset)`

Maps a previously opened file into memory, so that you can treat the contents of the file as any other pointer. The parameters are

- `addr`: a hint about where in memory you would like the file to be mapped. (Should almost always be `NULL`.)
- `length`: how many bytes of the file should be mapped into memory. (Typically this would be the file size.)
- `prot`: the protection mode of the mapped memory. For writing to a file, use `PROT_WRITE`.
- `flags`: what relationship the mapped memory should have to the original file. Normally `MAP_SHARED` is the correct choice, meaning that whatever is in the file will be reflected in memory (and vice-versa).
- `fd`: the file descriptor of the file to map into memory.
- `offset`: the offset within the file of the mapped memory. (Normally this will be 0.)

Assuming the call to `mmap()` succeeds, then afterwards you can access the contents of the file through the pointer that it returns. If it fails, then it returns `MAP_FAILED`.

`void perror(char *s)`

Prints a user-readable version of the last error that occurred when calling a system function. The string parameter `s` is printed before the error as a means of saying which component of the program caused the error; normally we just use `NULL` to print just the error message itself.

Since C doesn't have exceptions, it is important to check the return value after every function that could fail (which includes all of those above!). If the return value indicates an error, then use `perror()` to print an error message so you have some idea of what the problem was.

`map_file_for_writing()`

This will be similar to `map_file_for_reading()` in most ways, with a couple differences:

- Since we're writing now, we may need to create the destination file. (This is handled automatically for us by using `O_CREAT` in the call to `open()`.)
- Whether or not the destination file exists, we need to make sure that it is the correct size for the bitmap we'll write to it. (This is the purpose of the `file_size` parameter.) We'll do this using `ftruncate()`.

Similar to `map_file_for_reading()`, if any errors happen along the way (for instance, trying to write to a file that the user doesn't have permission on), the function should print an error message and return `NULL`.

The steps needed are the following:

A. Use `open()` to open the file for writing. The result is a file descriptor (`fd`).

B. Use `ftruncate()` to set the size of the file.

C. Call `mmap()` to map the file into memory.

D. Close the file using `close()`. (Once the file is mapped into memory, it is safe to close.)

E. Return the pointer returned from `mmap()`.

Most of this code will look almost the same as `map_file_for_reading()` so you might want to use it as a starting point. Please note that `open()` will need a third argument now (check the list of functions above for details) and that you will need to use `ftruncate()` in place of `fstat()`.

As before, make sure you check the return value of every function (except `close()`) for errors. Since C does not have exceptions, it is the responsibility of the programmer to write the code to check these things! If an error does occur, your function should first use `perror()` to print an error message, then return `NULL`.

Once you have `map_file_for_writing()` filled in, run it against the step 3 tester:

```
make test3
```

If you did it correctly, at this point it will give you a lot of complaints about `write_bitmap()` but not `map_file_for_writing()`.

`write_bitmap()`

Now fill in the `write_bitmap()` function. This will be fairly similar to `read_bitmap()` but you will be saving values into the file (using the `struct bitmap` parameter) instead of reading from it.

Here is the .bmp file format table again:

	Offset	Bytes	Field name	Notes
➡	0	2	Magic	Should always be "BM"
➡	2	4	File size	The total size of the file in bytes
	6	4	Reserved	Safe to ignore
➡	10	4	Pixel data offset	The location in the file where the pixel data begins
➡	14	4	Header size	Acts as a version number for the .bmp file
➡	18	4	Bitmap width	Width of bitmap in pixels
➡	22	4	Bitmap height	Height of bitmap in pixels
➡	26	2	Color planes	Should always be 1

	Offset	Bytes	Field name	Notes
➡	28	2	Color depth	Color depth (bits per pixel). For us, should always be 24.
➡	30	4	Compression method	We will not deal with compressed bitmaps, so should always be 0
➡	34	4	Image size	Size (bytes) of raw image data
	38	4	Horizontal resolution	Resolution in pixels per meter (safe to ignore)
	42	4	Vertical resolution	
➡	46	4	Palette size	Number of colors in palette
	50	4	Number of important colors	How many colors are "important" (ignore)
➡	*(variable)*	*(variable)*	Pixel data	For the files we work with, a sequence of `BB GG RR` values

The marked rows in the table are fields that your code must set for credit. Some notes on these:

- Any field consisting of multiple bytes (in particular, all of the marked ones) will need to be set using the same sort of syntax as you used in `read_bitmap()`. That is, the `byte *` for the file will need to cast to a pointer of another type (e.g., `int *` for the file size since it's four bytes) before you assign the field value.

  If you don't do this, the code will compile and seem to work, but it will fail the tester as it will assign only a single byte for the field.

- File size: this should be the total size of the file in bytes. The `bmp_file_size()` function provided in `project2.c` can calculate it for you.

- Pixel data offset: you should place the pixel data immediately after the headers (right after the number of important colors field), so the pixel data offset will always be **54** for our purposes.

- Header size: this should be **40** bytes.

- Image size: this is the total size of the pixel data portion of the file (not including headers). This is just the height of the bitmap times the stride.

- Palette size: we are not using a palette, so this should be 0.

The pixel data is stored in the same manner as from Step 2. Your `pixel_to_rgb()` function should be helpful in grabbing the BB, GG, and RR values for each pixel.

You might use something like the following for an overall structure for copying the pixel data.

```
// Compute the stride (number of bytes in one row)
int stride = (24 * width + 31) / 32 * 4;

for (int y = 0; y < bmp->height; ++y)
{
    // Calculate where this row of pixels begins in the file
    // and assign it to the "row" variable
    // (Use the pixel data offset and stride)
    // Remember that the file stores rows from bottom to top!
    byte *row = ...
```

```
            for (int x = 0; x < bmp->width; ++x)
            {
                // Get the pixel at (x, y) from bmp->pixels
                int pixel = ...

                // Decompose pixel into R, G, and B by using pixel_to_rgb(


                // Place the B, G, and R values into the appropriate locat
                // "row"

            }
        }
```

As always, run your code against the tester (`make test3`) to check it.

Running the application

Once you have the above working — or even before if you would like to do it to help debug your code — you should uncomment the call to `do_main_menu()` in your `main()` function. This function presents a menu-based interface for the user, letting them perform various operations; at this point the only available functions are reading an image and then saving it to another file, but you should test this on a few images (the `/var/project02/images` directory has several suitable files) to see that it works.

A program called `tiv` can be used to get a quick preview of an image (`tiv` ***`filename.bmp`***), but it may or may not work depending on the terminal you are using. You can always use `scp` to download an image from the server to look at it in more detail, of course! (I *believe* BMP images can be viewed natively on all major operating systems, but if I am mistaken, there are free web viewers such as Jumpshare's.)

Common errors

If you are having issues with this part, please check whether it matches any of the following. This section will continue to be updated as students ask the instructor for assistance.

→ **Tester fails with "Wrong file size: expected 438 but got *[some number]* (are you using bmp_file_size()?)"**
This often happens when you are not following the first note under the table of fields: you must cast the file's `byte *` to a different kind of pointer before assigning field values, or only one byte of the field will actually be placed.

End of first part

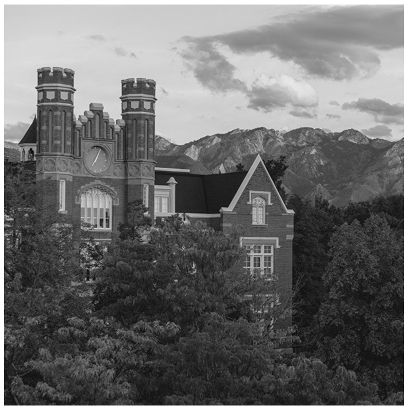
You have reached the end of the first deadline for Project 2. When you have completed all of the above steps, please submit them on Canvas. If you are feeling good about your progress so far, you may go ahead and get started on the remaining two steps!

Step 4

Goal Write functions that transform bitmap images without changing their dimensions (that is, just altering the colors).

Converting to grayscale

Original


Transformed

Now that we can access an image pixel by pixel, we can work on transforming some images. Our first transformation is to make each pixel *grayscale*: black, white, or a shade of gray. In terms of RGB colors, this means that within each pixel, R = G = B.

In your `project2.c`, go to the function `bitmap_to_grayscale(struct bitmap *bmp)`. The goal of this function is to replace each pixel with its grayscale equivalent (that is, replace R, G, and B with their average, (R + G + B) / 3). Use your `pixel_to_rgb()` function to pull out the R, G, and B of each pixel, average them, and then recombine them using `rgb_to_pixel()`.

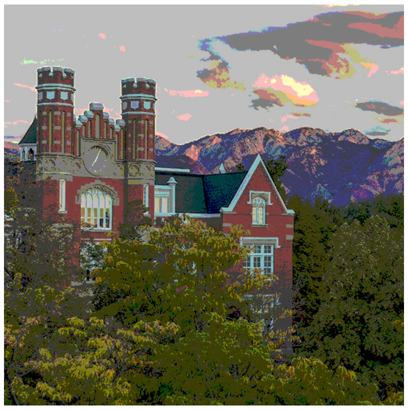
Note that it is very important that your `bitmap_to_grayscale()` function *only* converts the bitmap to grayscale, doing *nothing* else. It should not read or write a file, print out anything, take input from the user, or anything of the kind. This is important so that it integrates cleanly with the application's main menu.

You can test your `bitmap_to_grayscale()` function by enabling the call to `do_main_menu()` in your `main()` and choosing the grayscale option from the menu, then saving the result to a file. You can use `tiv` at the command line to check that the image has been rendered grayscale, or use `scp` to transfer the file to your local computer.

You can also use the Step 4/5 tester (`make test4_5`) to test your code against several reference images. (It will throw up errors about the other three functions these steps until they are filled in, of course.)

Posterizing

Original


Transformed

Posterization is the process of reducing the number of colors present in an image, making it more economical to print (perhaps on posters, which is where the name comes from). There are a number of ways to do this, but for purposes of this project, we will apply the following process to each of the R, G, and B components of each pixel:

Original R, G, or B value	Final R, G, or B value
Less than 32	0
32 to 95	64
96 to 159	128
160 to 223	192
224 or more	255

By doing this, there are 5×5×5 = 125 possible colors, rather than the full gamut of 256^3 = 16,777,216 possible 24-bit color values.

Fill in the `bitmap_posterize(struct bitmap *bmp)` function in your `project2.c` so that it posterizes all the pixels in the image. You can test it in the same way that you tested `bitmap_to_grayscale()`. (The `/var/project02/image_outputs` directory has samples of what the test images should look like after posterization.)

Step 5

Goal Write functions that transform bitmap images in ways that alter the image dimensions (e.g. stretching).

Resizing images

Operations that change the image size are a little more complicated because the size of the image data is changing, so you will need to allocate new memory for the bitmap's pixels. I recommend the following overall structure for doing this:

A. Use the original width and height of the bitmap to calculate `new_width` and `new_height`.

B. Call `malloc()` to allocate enough memory for the pixels of a `new_width`×`new_height` image (call this `new_pixels`).

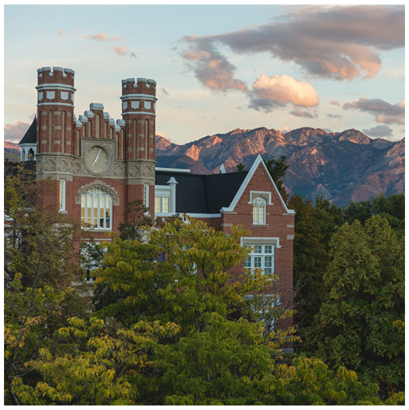
C. Fill in the pixel data of `new_pixels` using a nested `for` loop like you did before. Depending on which operation you are doing, it might be more convenient to think of this as iterating over the original image (width×height) or over the new one (`new_width`×`new_height`).

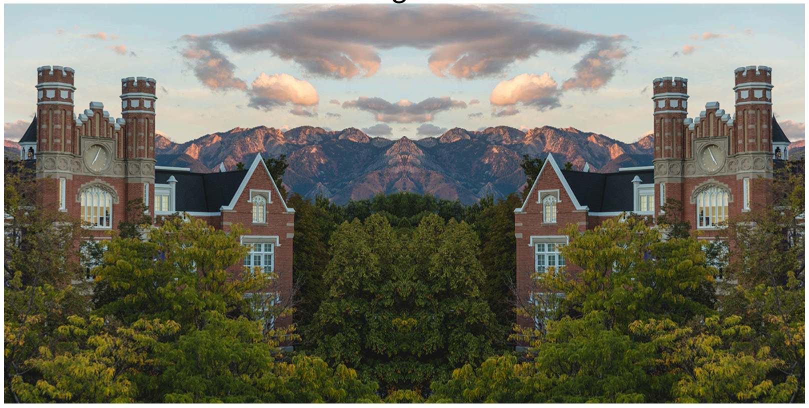
D. `free()` the old pixel data now that it's no longer needed.

E. Update the width, height, and pixel data of the bitmap image to their new values (`new_width`, `new_height`, `new_pixels`).

Use this structure for implementing both of the operations below.

Mirroring


Original


Transformed

The result of this operation is a bitmap image that is the same height as the original but twice the width. If the original image is 500×500, the mirrored version will be 1000×500; the original pixel at the top left (0, 0) will be at both the top left (0, 0) and top right (999, 0) of the mirrored version, and so on.

Fill in the function `void bitmap_mirror(struct bitmap *bmp)` so that it performs the mirroring operation on the given bitmap. After the call, whatever bitmap `bmp` points to should have twice the width that it did before and its `pixels` should point to the new pixel data; **make sure to `free()` the original pixel data in the `bitmap_mirror()` function to prevent a memory leak** (but you should not do this until after you have done the mirroring operation, because until then you still need the original pixels!).

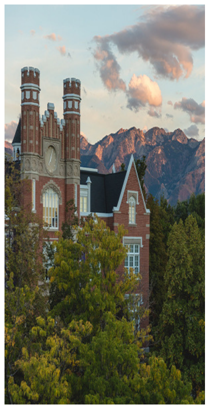
Some hints on doing this:

- Follow the overall structure shown above. For this operation, it is probably more convenient to iterate over the original (width×height) image.
- You do not need to use the `pixel_to_rgb()` or `rgb_to_pixel()` functions here; just copy the pixel `int` values straight over.
- Be very careful with how you index both the `pixels` and `new_pixels` arrays — make sure that you are using the right values in the `y * width + x` formula for each!
- Also be careful that you are calculating the mirrored locations of pixels correctly: taking the example of a 500×500 image again, the original (0, 0) pixel should be present at both (0, 0) and at (999, 0) in the mirrored image. This is a case where it's easy to make off-by-one errors, so I recommend writing down what you think the calculation of the mirrored pixel should be and working a few test cases out on paper to make sure they come out right.
- Don't forget to `free()` the original pixel data to prevent a memory leak!

As before, you can test your function by running the `step4_5` tester, or by running the menu-based application.

Squashing

Original


Transformed

Squash the image vertically so that its height is unchanged but the width is halved. Do this by taking each adjacent pair of pixels in the original image, averaging their red, green, and blue values, and using that to form a single pixel in the new image.

For example, if the original image is 500×500, then the squashed image will be 250×500. The two top-left pixels (0, 0) and (1, 0) will have their red, green, and blue averaged to form the new top-left pixel at (0, 0); to the right of that, (2, 0) and (3, 0) will be averaged to form the new (1, 0). The last pixels in the top row (498, 0) and (499, 0) will be averaged to form the pixel at (249, 0), and so on.

Fill in the function `void bitmap_squash(struct bitmap *bmp)` to perform this operation. Make sure it doesn't leak memory (`free()` the original pixel data when done with it!).

Some hints:

- It is probably more convenient to iterate over the new (`new_width`×`new_height`) image.
- Make sure you average the red, green, and blue components *separately*. Averaging the overall pixels themselves is an interesting effect but not what we want!
- If the original width is odd, then you don't need to worry about that that last column of pixels on the far right. (If the original image is 501×500, then the squashed image will be 250×500 and the last pixel in each row of the original image is just dropped.)
- Don't forget to `free()`!

Test your function in the usual way.

Extra credit

For up to 10 points of extra credit, make all of the following additions to your program.

Fix the memory leaks from mapping files into memory

Right now, the memory allocated by the `mmap()` calls in `map_file_for_reading()` and `map_file_for_writing()` is never released. In particular, if the user chooses to save a file repeatedly, eventually the program will run out of memory and be killed by the operating system. (This could even happen very soon if they input a large image!).

However, memory allocated by `mmap()` cannot be passed to `free()` (remember that `free()` only works with memory allocated by `malloc()` and friends — `mmap()` is something different). Instead, use the following function:

```
int munmap(void *addr, long length)
```

Deallocates memory previously allocated using `mmap()`. The parameters are

- `addr`: the pointer previously returned by `mmap()`.
- `length`: the length of the file previously allocated. (Must match the `length` previously passed to `mmap()`.)

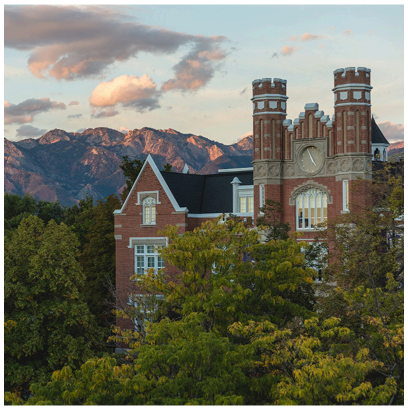
You will need to call `munmap()` appropriately in `main()` after any calls to `read_bitmap()` and `write_bitmap()`.

Add four additional image operations

For full credit, each one needs to be implemented in a function in the same way as the previous four operations (e.g. `void bitmap_reflect(struct bitmap *bmp)` for the reflect operation) **and must not leak memory**. (Be sure to `free()` the old `pixels` pointer when done with it!)

Reflecting

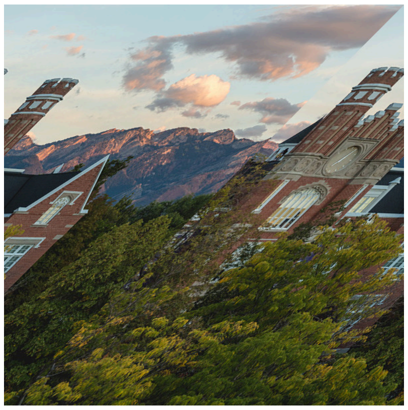
Original


Transformed

Reflects the image horizontally (so what was originally on the right side of the image is now on the left, and vice-versa). The size of the image is unchanged.

Skewing

Original

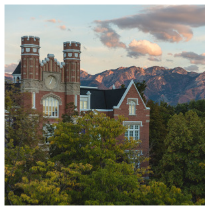

Transformed

Skews the image, moving each row over relative to the one above and wrapping around. The size of the image is unchanged.

- The top row is the same as the original.
- The second row is moved to the left by 1 pixel. (The rightmost pixel is what used to be on the far left end.)
- The third row is moved to the left by 2 pixels. (The rightmost two pixels used to be at the far left end.)
- The fourth row is moved to the left by 3 pixels. (The rightmost three pixels used to be at the far left end.)
- And so on. If the image is square (like the Converse Hall example shown above) then the bottom row will be the same as in the original image.

Shrinking

Original

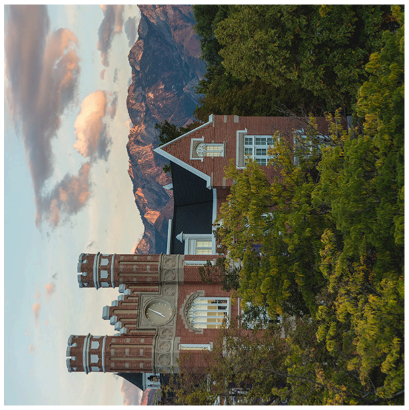

Transformed

Squashes the image both vertically and horizontally at the same time. Each pixel in the result is obtained by averaging the red, green, and blue components of a 2×2 square of pixels from the original image. The final image has half the width and half the height of the original.

As in the case of the squash operation, if the width and/or height of the image is odd, then there is a row and/or column of the original image that is disregarded.

Rotating

Original


Transformed

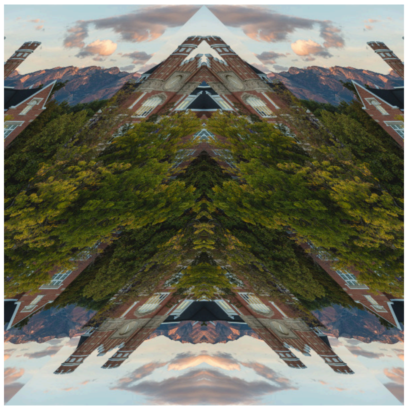
Rotates the image left by 90°. The width and height of the image are swapped (if the original image had dimensions A×B, then the rotated image will be B×A).

Add the new operations into the menu

Make the new operations selectable from the menu in the same way as the basic four operations. As before, the user should be able to choose multiple operations in sequence to perform on the image. For example, the following is the result from performing the following sequence of operations:

Rotate, mirror, rotate, rotate, rotate, skew, mirror, shrink

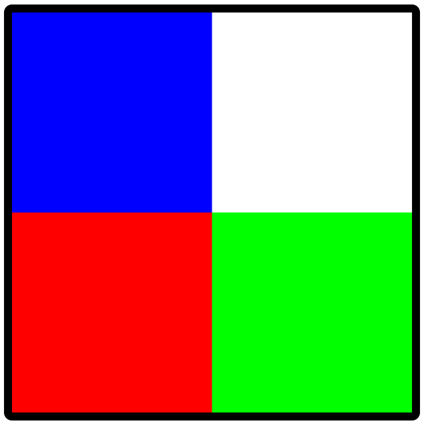
Original


Transformed

Pixel orders

.bmp file pixel order

In a .bmp file, the pixels are stored in rows (each row being a stride of bytes as above); the rows are stored in a bottom-up fashion in the file: the first row of pixels in the file is the *bottom* row and the last is the *top* row. Suppose the image in question is the 2×2 image shown (greatly expanded) below:



Then the pixel data in the .bmp file would look like this:

```
□ 00 00 FF □ 00 FF 00 00 00        (bottom row, left to right, plus padding)
□ FF 00 00 □ FF FF FF 00 00        (top row, left to right, plus padding)
```

However, when the image data is saved into the `struct bitmap`, we want to store the pixels in the usual order for computer images: left-to-right, top-to-bottom (English reading order). That is, we would want to store these same pixels as the array

```
{ □ 0x0000FF, □ 0xFFFFFF, □ 0xFF0000, □ 0x00FF00 }
```

.bmp file stride

If a .bmp image has width × height pixels, each of which has three bytes `BB GG RR`, you might expect the total size of the image data to be width × height × 3 bytes, but unfortunately this isn't *quite* right. In a .bmp file, every row of the image (horizontal line of pixels) needs to be a multiple of exactly four bytes long. Suppose the image is just 2×2 pixels: then the pixel data would look like

```
BB GG RR BB GG RR 00 00
BB GG RR BB GG RR 00 00
```

where the **`00`**s are inserted to make each row a multiple of four bytes long. Each row has two pixels (6 bytes) plus 2 bytes of padding to make 8 bytes. The total number of bytes in one row of pixels is called an image's *stride* (8 bytes in this example). The stride can be calculated as

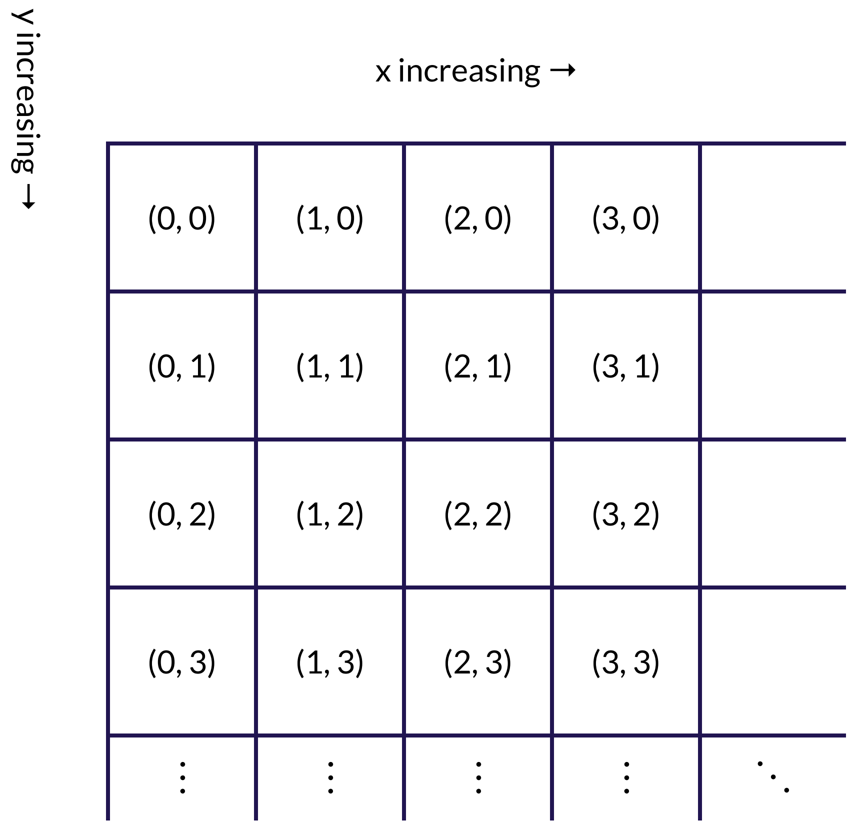
```
stride = (24 * width + 31) / 32 * 4;
```

You will need to use the stride in your code when you are mapping rows of the file data into the actual pixels in the program's copy of the image.

Mapping (x, y) coordinates to an index

As previously stated, we are storing the image pixel data in our `struct bitmap` in English reading order: the first pixel is the top left one, the second is the pixel to its right, and so on for the first row. After the first row, we store the second row in the same way (with no padding between), repeating to the last (bottom) row of the image. In other words, the order of pixels in the `struct bitmap` is upside-down from in the .bmp file: in the file, the first row is the *bottom*, while in the `struct bitmap`, the first row is the *top*.

Since you will need to place pixel data by coordinates, it is important to know how to map an (`x`, `y`) point in the image to an index `i` in the pixel data. Consider the following map showing the top left corner of an image:



In memory (in the `struct bitmap`'s `pixels` array) these pixels would be stored in the following order:

Index	`0`	`1`	`2`	`3`		`width`	`width+1`	`width+2`	`width+3`	
Pixel	(0, 0)	(1, 0)	(2, 0)	(3, 0)		(0, 1)	(1, 1)	(2, 1)	(3, 1)	

Mapping an (`x`, `y`) point in the image to an index `i` in the array is not difficult: the formula is

```
i = y * width + x
```

where `width` is the width of the image in pixels. Any time you need to find a given pixel in the array by its (`x`, `y`) coordinates, you can use this calculation.

Debugging

Since this project is relatively complex, it is helpful to use a debugger when you run into steps. You should already know how to use a (graphical) debugger, such as the one built into IntelliJ IDEA or JGrasp, but there are also text-mode debuggers for use on the command line, such as `gdb`, the GNU debugger (the `g` in `gdb` is the same `g` as in `gcc`).

Debug symbols

In order to use a debugger on your compiled code, is is very important that you include debug symbols when compiling. This is easy: you just need to remember to add `-g` to the command line when compiling, for example, `gcc -g project2.c -o project2`. The Makefile supplied to you for the project has this feature builtin; all you need to do is use `make debug` instead of just regular `make`, for example

- `make debug project2` to compile your project2;
- `make debug test1` to run the step 1 tester with debug symbols (also automatically runs the debugger for you).

Using gdb

Once you have your program compiled with `-g`, you run it in the debugger by using the `gdb` command:

```
gcc -g project2.c -o project2
gdb project2
```

(If you are using one of the project testers with `make debug` then it will automatically run the debugger for you.)

The debugger has the same commands as a graphical debugger, accessed by typing commands. (Most of them can be abbreviated for ease of typing.)

- `run` (or `r`): run your program. It will pause if it hits a breakpoint or an error (such as a segmentation fault) occurs.
- `break` (or `b`): set a breakpoint. You can do this either by *filename:line#* or by the name of a function, for example

  ```
  break project2.c:298          # set a breakpoint on line 298 of project2.c
  b map_file_for_reading        # set a breakpoint at the beginning of map_file
  ```

- `print` (or `p`): print a value, such as a variable, an array element, or an expression. For example,

  ```
  print x                       # prints the value of variable x
  p file[j * stride + i];       # prints that element in the array
  ```

- `next` (or `n`): execute the next statement. (If that statement has a function call in it, this *will not* go inside it — it is a “step over.”)
- `step` (or `s`): step into the next instruction. (If that statement has a function call in it, this *will* go inside it — it is a “step into.”)
- `continue` (or `c`): continue executing the program (until another breakpoint is hit or error occurs).
- `backtrace` (or `bt`): shows all the functions in the current call stack with their parameters. Here's an example:

  ```
  (gdb) bt
  #0  solve_quadratic (a=1, b=0, c=-1, x1=0x7fffffffe360, x2=0x7fffffffe358)
  #1  0x00005555555551a2 in main () at debugging.c:15
  ```

  This means that execution is currently at line 27 in the `solve_quadratic()` function (the top of the stack), which has the listed values for parameters `a`, `b`, `c`, `x1`, and `x2`, and this function was called from `main()` line 15.

- `quit` (or `q`): quits the debugger.

If your program gets stuck in an infinite loop, you can type Ctrl-C to cancel execution.