CSE 220: Systems Fundamentals I

Stony Brook University
Homework Assignment #2
Spring 2024
Due: Tuesday, March 5th, 2024 by 9:00 pm EST

Updates to this Document

- None yet!

Learning Outcomes

By the end of this assignment you should be able to do the following in C:

- Parse command line arguments from `argv[]`.
- Open, read, write and close text files.
- Use functions from `stdio.h` to perform file I/O operations.
- Manipulate 2D arrays.

Change in CodeGrade

To discourage last-minute pounding on the CodeGrade servers and to encourage you to develop in CodeSpaces or locally, CodeGrade will accept submissions only once every 5 minutes from you. If you push your code to GitHub more frequently than that, CodeGrade will ignore the pushes.

Change in Compiler

Unfortunately, the version of valgrind that we have access to on Codespaces is too old to work with the newer versions of clang. I spent hours trying to get the newest version of valgrind to build and work correctly with clang 14 with no luck. Therefore, I am going to switch us back to gcc, which will hopefully resolve valgrind crashes that some of you are experiencing in Codespaces.

Note that when you start up Codespaces and it asks you to select a C compiler kit, **please select gcc** instead of clang.

Assignment Overview

You will write a non-interactive, command-line program that reads an image file from disk, performs image manipulation operations given as command-line arguments, and writes the modified image back to disk. We imagine an image in memory as a 2D **raster** (grid or array) of

pixels in which the upper-left corner has position <row #0, column #0> and the lower-right corner has position <row #(H-1), column #(W-1)>, where H is the number of rows in the image (i.e., the image's height) and W is the number of columns in the image (i.e., the image's width). Note that we are **not** using (x, y) coordinates to reference pixels, but rather row and column numbers.

You will be working with image data provided in a home-grown file format which we will simply refer to as "SBU file format". Image files will end with the extension `sbu` (e.g., `face.sbu`). You will also work with a standard format called PPM (portable pixmap). PPM file names end with `.ppm`.

PPM and SBU File Formats

Before detailing the structure of an SBU image file, let's look at a simpler format, called PPM . PPM format (specifically, the P3 format variant) stores image data in plaintext. The header of the file consists of the characters "P3", followed by the integer width of the image (in pixels), the integer of the height (in pixels), and the maximum value for a color component. We will assume that this value is always 255 (the maximum value that can be stored in an `unsigned char`). After the header is the color information for each pixel. A pixel's color is represented by three integers each in the range 0–255 denoting the red, green, and blue components (in that order).

For example, the 100x72 image file `tests/images/seawolf.ppm` begins as follows:

```
P3
100 72
255
0 0 0 0 0 0 0 0 0 …        ←     R0 G0 B0 R1 G1 B1 R2 G2 B2 …
```

Each (R, G, B) color component is separated from the next by a space or newline. Newlines do not necessarily indicate the end of a row of pixels.

A very handy VS Code extension has been pre-installed in the CodeSpaces Docker image for you called "PBM/PPM/PGM Viewer for Visual Studio Code". You can use the extension to view your program's PPM output.

To see the ASCII character contents of a PPM file, type a command similar to the following in the terminal:

```
less tests/images/seawolf.ppm
```

Now, an SBU image file contains two sections:

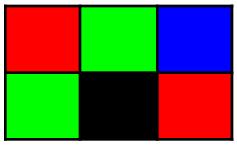
- a color table, which maps an unsigned integer to an RGB (red, green, blue) triple; and

- pixel data, some of which could be compressed using run-length encoding

It is structured similarly to a PPM file. It begins with the characters “SBU” on a line by themselves, followed by the width and height in pixels, separated by one space and on a line by themselves. After the dimensions come the number of entries in the color table, followed by the color table entries themselves, which consist simply of RGB triples, all on a single line. As an example, a two-entry table with color #0 as (255, 128, 128) and color #1 as (55, 66, 77) would be the single line:

```
2 255 128 128 55 66 77
```

After the color table we find the pixel data. Single pixels are represented by their index in the color table. For example, the 3x2 image below:



would be represented as the following file:

```
SBU
3 2
4 255 0 0 0 255 0 0 0 255 0 0 0
0 1 2 1 3 0
```

Note that the representation of an image is not unique because the colors in the table could be reassigned to different positions in the table. We will deal with this issue later when we prescribe the algorithm you must use to generate the color table.

For images containing runs of two or more pixels with identical colors, which could even spill over from one row to the next, we do not represent each pixel separately. Rather, we encode these runs of identical pixels using run-length encoding:

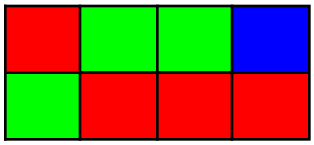
```
*count table_index
```

For example, `*8 5` encodes a run of 8 pixels, all with color #5.

When building the color table to save an image in SBU format, scan the pixels starting at the upper-left corner, at row 0, column 0. Scan the pixels one row at a time (left-to-right), then top-to-bottom, inserting each new color into the table. The first color found must be stored at index 0, the second color at index 1, etc.

When encoding the pixels in a file, only runs of two or more pixels should be encoded using the RLE notation described above. Single pixels should be represented as a single index value, not, for example, as `*1 15`. Save the pixels in the same order they would be in a PPM file: row-by-row (left-to-right), then top-to-bottom order.

For example, the 4x2 image below:



would be represented as the following file:

```
SBU
4 2
3 255 0 0 0 255 0 0 0 255
0 *2 1 2 1 *3 0
```

A Quick Primer on File I/O

Reading and writing files is not very different from using `scanf` and `printf` when the files have simple and predictable formats.

To open a file for **reading** `("r")`,we can type:

```
FILE *fp = fopen("./dir1/dir2/data.txt", "r");
```

`fp` is called a **file handle** and provides a means to read from the file. For example:

```
unsigned int i, j;
fscanf(fp, "%u %u", &i, &j);
```

will read two integers (separated with a space) from the file referenced by the `fp` file handle. `fscanf` will helpfully skip over whitespace while looking for the input values you specified. There is no need to search for newline characters and discard them, for instance, so refrain from mixing use of `fscanf` and `fgetc`.

Be sure to close the file when you have finished reading from it:

```
fclose(fp);
```

To write data to a file, first open it for **writing** (`"w"`). The code below will create the file if it does not exist. If the file does exist, it will be overwritten:

```
unsigned int width = 200;
FILE *fp = fopen("./output/results.txt", "w");
fprintf(fp, "%u", width);
```

Again, remember to call `fclose` when done writing.

If a file cannot be successfully opened for reading or writing, the call to `fopen` will return `NULL`. For example:

```
FILE *fp;
if ((fp = fopen(FILE_NAME, "r")) == NULL)
    // code to handle the failure
```

Template Repository

Starter code is available on CodeGrade (and GitHub), as with previous assignments.

Part 1: Command-line Argument Validation

The program must check for the error situations given below, which have been provided in decreasing order of priority. That is, the first error case has the highest priority. You should check for the errors listed below in the order shown. If there are no errors with the command-line arguments, return 0.

As you read through Part 1, you should refer to Part 2 to see what the various command-line arguments mean (`-i`, `-o`, `-c`, `-p`, `-r`).

At a minimum, the program must be provided paths to the input and output image files. These are given by the `-i` and `-o` arguments, respectively. For example:

```
./build/hw2_main -i "./images/face.sbu" -o "./output/moon.sbu"
```

Error Code (see `hw2.h`)	**Explanation**
`MISSING_ARGUMENT`	At least one of the `-i` and `-o` arguments is missing; or, a command-line argument is given, but no parameter (e.g., filename) for the argument is given. See the examples in `./tests/src/tests_validate_args.cpp`. One way to check if the parameter is missing is to see if the

	token after an argument begins with a “-” character. We will assume that this indicates the parameter is missing.
`UNRECOGNIZED_ARGUMENT`	An unrecognized argument was provided on the command line. See Part 2 for the list of valid arguments.
`DUPLICATE_ARGUMENT`	A recognized argument is provided two or more times.
`INPUT_FILE_MISSING`	The named input file is missing or cannot be opened for reading for some reason. You may assume that if the input file exists, it will always end in .ppm or .sbu.
`OUTPUT_FILE_UNWRITABLE`	We cannot create the output file for writing for some reason.
`C_ARGUMENT_MISSING`	The `-p` argument has been provided at least once, but the `-c` argument was not provided.
`C_ARGUMENT_INVALID`	Too many or too few parameters have been provided to the `-c` argument.
`P_ARGUMENT_INVALID`	Too many or too few parameters have been provided to the `-p` argument.
`R_ARGUMENT_INVALID`	Too many or too few parameters have been provided to the `-r` argument, or the indicated font file could not be opened. Assume that if the font file exists, then its contents are valid.

When returning these error codes, use the macros that have been `#define`’d in `hw2.h`. (e.g., write `return MISSING_ARGUMENT;`, not `return 1;`) We will probably change the definitions of these macros during grading.

To make it easier to parse command-line arguments, use the function `getopt`. Check out this tutorial on command-line argument parsing written by Prof. Paul Krzyzanowski at Rutgers University. It is well-written and gives you concrete examples and sage advice on how to use `getopt`. If you choose not to use `getopt` for the assignment, you are still responsible for having a basic understanding of how to use it.

Consider using the functions `strtol` or `strtoul` to help you parse integers present in the arguments. `strtoul` makes a best-effort attempt to parse its input to a `long`. A string like `"53sbu"` will be converted into the `unsigned long 53`, which your program should accept as a valid integer. When `strtoul` totally fails to convert a string into a `long`, it returns 0. For some of the command line arguments, 0 is a valid input value, and your code should accept it as valid.

For many examples of how errors in the command-line arguments should be handled, refer to the file `./tests/src/tests_validate_args.cpp`. More test cases might be added after the assignment's release in response to students' questions.

Part 2: Functionality and Command-line Arguments to Implement

All communication with your program will be through command-line arguments and files that are saved and loaded. It is up to you to choose how to appropriately decompose your program into reusable functions.

`-i`:Path to input file

The `-i` argument provides the path to the image file to load. Your program must be able to read PPM and SBU files. The file extension should be used to determine the format of the input file. The file path could be a relative path or an absolute path, so your program must not manipulate the string containing the path or assume that it will always be in a particular directory. The `-i` argument must appear exactly once in the command-line arguments.

Command-line Arguments	**Explanation**
`-i "./images/face.sbu"`	Load the `face.sbu` image file stored in the local `images` directory.
`-i "/home/testing/moon.ppm"`	Load the `moon.ppm` image file stored in the `/home/testing` directory.

`-o`: Path to output file

The `-o` argument provides the path where the processed image file must be saved. Your program must be able to read PPM and SBU files. The file extension should be used to determine the format of the output file. The file path could be a relative path or an absolute path, so your program must not manipulate the string containing the path or assume that it will always be in a particular directory. The `-o` argument must appear exactly once in the command-line arguments.

Command-line Arguments	**Explanation**
`-o "./images/baby.ppm"`	Save the processed image in a file named `baby.ppm` in the local `images` directory.
`-o "/home/testing/sun.sbu"`	Save the processed image in a file named `sun.sbu` in the local `/home/testing` directory.

`-c`: Rectangular region to copy pixels from

The `-c` argument is used to copy a region of pixels that can later be pasted into a different part of the image. It requires four *parameters* separated by commas: `row`, `col`, `width` and `height`:

- `row`: the row number of the upper-left pixel of the region (0-based indexing)
- `col`: the column number of the upper-left pixel of the region (0-based indexing)
- `width`: the width of the entire region, in pixels (i.e., the number of columns spanned)
- `height`: the height of the entire region, in pixels (i.e., the number of rows spanned)

You may assume that at least one pixel of the selected region actually falls within the bounds of the image. The `-c` argument may appear at most once in the command-line arguments. Hint: to split the argument into the four components separated commas, consider using the `strtok` function.

Example

Command-line Arguments	Explanation
`-c 5,8,20,30`	Select the region starting at the pixel at row 5, column 8 that is 20 columns wide and 30 rows high. The lower-right corner of this region would be at the pixel at row 34, column 27.

`-p`: Paste the copied contents into the image

Assuming that the `-c` argument has been validly specified, the `-p` argument indicates where in the image the selected (copied) pixels should be pasted. Position the upper-left corner of the copied region at row `row` and column `col` of the image and then overwrite the contents of the image with the selected region's pixels. Any pasted pixels that fall outside the boundary of the image should simply be discarded. The row and column *parameters* are separated by a comma in the command line.

- `row`: the row number of the upper-left pixel of the image where the copied region's upper-left corner should be positioned for pasting (0-based indexing)
- `col`: the column number of the upper-left pixel of the image where the copied region's upper-left corner should be positioned for pasting (0-based indexing)

The `-p` argument may appear at most once in the command-line arguments.

Examples

Command-line Arguments	Explanation

`-p 19,50`	Paste the contents of the copied region starting at row 19, column 50, overwriting the pixels at the region covered by the pasted content.

Below is an example of a copy-and-paste operation on the file `desert.ppm` (286x200).

Command line arguments: `-c 90,10,50,100 -p 90,60`

Before **After**

`-r`: Print a message on top of the image

This command prints a message on top of the image in white (RGB = (255,255,255)), with the upper-left corner of the first letter's bounding box positioned at the given row and column in the image. The `-r` argument requires the following *parameters*, which are separated by commas.

- `message`: the message to display, given in double quotation marks.
- `path_to_font`: the path to the file containing the font file, given in double quotation marks.
- `font_size`: the integer multiplier used to increase the size of the text horizontally and vertically
- `row`: the row number of the upper-left pixel of the image where the upper-left corner of the first letter's bounding box should be positioned when printing the message into the image
- `col`: the column number of the upper-left pixel of the image where the upper-left corner of the first letter's bounding box should be positioned when printing the message into the image

The `r` argument may appear at most once in the command-line arguments.

You have been provided a set of text files in the `fonts` directory that define some simple fonts. Looking closely at the files, each letter is defined as a group of asterisks and spaces, and is separated from the neighboring letter by a single column of spaces. We define the **bounding box** of a letter as the smallest rectangular grid of asterisks and space characters that enclose the letter. For example, the bounding box for the letter “G” in `fonts/font1.txt` is as shown below, where the spaces are visualized using periods for the sake of understanding:

```
.******.
**......
**...***
**....**
.******.
```

When a letter is “printed” on top of an image at a given position, the pixels that correspond with asterisks are changed to white (RGB = (255,255,255)). Pixels that correspond with spaces in the font file are left unchanged. The images below show an image before and after printing the letters of "SEAWOLVES" with the command line arguments

```
-r "seawolves","fonts/font1.txt",1,10,5.
```

Note that the upper-left corner of S’s bounding box is located at row 10, column 5 of the image. Between adjacent letters in the image, one column of unmodified pixels has been skipped over so that the letters are not squished together. Do the same in your implementation

Before **After**

The only valid characters of a message are letters and spaces. You may assume that the message contains only valid characters. Any lowercase letters in the input string are converted to uppercase letters. Space characters are “printed” by leaving five columns of unmodified pixels. If a message runs off the screen horizontally or vertically, only modify pixels of the image for letters that can be completely displayed. In other words, if a letter’s bounding box falls outside the image boundaries, do not display any portion of the letter.

Example: `-r "stONY brOOK","fonts/font2.txt",1,60,5`

Before **After**

Example: `-r "new YORK state","fonts/font3.txt",1,10,5` Note how a portion of the text is chopped off as the message runs off the image, but each individual letter is entirely printed or omitted.

Before

After

The `font_size` argument simply scales each character by the provided integer. Assume that the valid range is 1 through 10, inclusive. A value of 1 for `font_size` indicates no scaling. Going back to the letter G example from before, below is shown the letter with `font_size` of 1, 2 and 3, respectively. Continue to leave only one space between adjacent letters, even when `font_size` is greater than 1. Similarly, a space character in the input message should still be rendered as five blank columns of pixels.

```
.******.      ..************..      ...******************...
**......      ..************..      ...******************...
**...***      ****............      ...******************...
**....**      ****............      ******..................
.******.      ****......******      ******..................
              ****......******      ******..................
              ****........****      ******.........*********
              ****........****      ******.........*********
              ..************..      ******.........*********
              ..************..      ******............******
                                    ******............******
                                    ******............******
                                    ...******************...
                                    ...******************...
                                    ...******************...
```

Examples:

```
-r "stony brook university","fonts/font4.txt",1,80,10
```

```
-r "stony brook university","fonts/font4.txt",2,80,10
-r "stony brook university","fonts/font4.txt",3,80,10
```

Before

After (size = 1)

After (size = 2)

After (size = 3)

Command Execution

Assuming that the command line arguments are valid, execute them in the following order:

1. Load the image file from disk into memory.
2. Copy the selected region (if requested in the command line arguments).
3. Execute the paste operation (if requested in the command line arguments).
4. Print the message (if requested in the command line arguments).
5. Save the image to disk.

Testing & Grading Notes

To test for the “equality” (more like “equivalence”) of two image files, the following procedure is followed:

1. The expected and actual files are opened for reading.
2. The `tr` utility replaces all of the newlines in each file with spaces, and then replaces each run of spaces with a single space. The result of this process is to change each file to one line. All values stored in the file will be separated by a single space.

3. The `diff` tool checks if the two modified files are the same.

More test cases (both visible and hidden) might be added after the assignment's release in response to students' questions.

Note: any evidence of hard-coding test case input or output will automatically result in a score of zero for the assignment. This kind of behavior is borderline academic dishonesty.

WARNING: Do not attempt to circumvent the testing framework by copying files out of the expected_outputs/ or images/ directories. Such behavior will be reported to the Academic Judiciary. The TAs will specifically look for this after autograding has completed.

Generative AI

Generative AI may not be used to complete this assignment.

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work for grading you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (algorithms, software design, text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the University's Academic Judiciary.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating on a homework assignment may be discovered and penalized after the grades for that homework have been recorded.

8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.