

image-processing

EECS 280 Project 2: Image Processing

Due 11:59pm ET Wednesday May 21st, 2025. You may work alone or with a partner ([partnership guidelines](#)).

Spring 2025 release.

Introduction

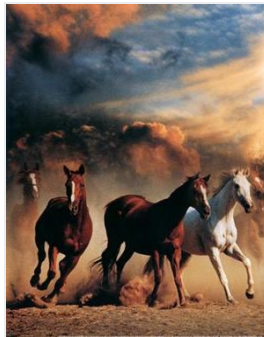
Build an image resizing program using a seam-carving algorithm.

The learning goals of this project include Testing, Debugging, Pointers, Strings, Streams, IO, and Abstract Data Types in C. You'll gain practice with C-style pointers and structs.

When you're done, you'll have a program that uses seam carving for content-aware resizing of images. The algorithm works by finding and removing "seams" in the image that pass through the least important pixels. For a quick introduction, check out [this video](#).



Original Image: 479x382



Resized: 300x382



Resized: 400x250

Setup

Set up your visual debugger and version control, then submit to the autograder.

Visual debugger

During setup, name your project `p2-image-processing`. Use this starter files link:

<https://eecs280staff.github.io/image-processing/starter-files.tar.gz>

[VS Code](#)[Visual Studio](#)[Xcode](#)

If you created a `main.cpp` while following the setup tutorial, rename it to `resize.cpp`. Otherwise, create a new file `resize.cpp`. You should end up with a folder with starter files that looks like this. You may have already renamed files like `Matrix.cpp.starter` to `Matrix.cpp`.

```
$ ls
Image.cpp.starter      dog_4x5.correct.ppm
Image.hpp              dog_cost_correct.txt
Image_public_test.cpp  dog_energy_correct.txt
Image_test_helpers.cpp dog_left.correct.ppm
Image_test_helpers.hpp dog_removed.correct.ppm
Image_tests.cpp.starter dog_right.correct.ppm
Makefile               dog_seam_correct.txt
Matrix.cpp.starter     horses.ppm
Matrix.hpp             horses_300x382.correct.ppm
Matrix_public_test.cpp horses_400x250.correct.ppm
Matrix_test_helpers.cpp horses_cost_correct.txt
Matrix_test_helpers.hpp horses_energy_correct.txt
Matrix_tests.cpp.starter horses_left.correct.ppm
crabster.ppm           horses_removed.correct.ppm
crabster_50x45.correct.ppm horses_right.correct.ppm
crabster_70x35.correct.ppm horses_seam_correct.txt
crabster_cost_correct.txt jpeg.hpp
crabster_energy_correct.txt processing.cpp.starter
crabster_left.correct.ppm processing.hpp
crabster_removed.correct.ppm processing_public_tests.cpp
crabster_right.correct.ppm resize.cpp
crabster_seam_correct.txt unit_test_framework.hpp
dog.ppm
```

Here's a short description of each starter file.

File	Description
<code>Matrix.hpp</code>	Interface specification for the <code>Matrix</code> module.
<code>Image.hpp</code>	Interface specification for the <code>Image</code> module.
<code>processing.hpp</code>	Specification of image processing functions that are pieces of the seam carving algorithm.
<code>Matrix.cpp.starter</code>	Starter code for the <code>Matrix</code> module.
<code>Image.cpp.starter</code>	Starter code for the <code>Image</code> module.

File	Description
<code>processing.cpp.starter</code>	Starter code for the <code>processing</code> module.
<code>Matrix_tests.cpp.starter</code>	Starter code for unit testing the <code>Matrix</code> module.
<code>Image_tests.cpp.starter</code>	Starter code for unit testing the <code>Image</code> module.
<code>Matrix_public_test.cpp</code>	Public tests for the <code>Matrix</code> module.
<code>Image_public_test.cpp</code>	Public tests for the <code>Image</code> module.
<code>processing_public_tests.cpp</code>	Tests for the <code>processing</code> module and seam carving algorithm.
<code>Matrix_test_helpers.hpp</code> <code>Matrix_test_helpers.cpp</code> <code>Image_test_helpers.hpp</code> <code>Image_test_helpers.cpp</code>	Helper functions for unit tests
<code>unit_test_framework.hpp</code>	A simple unit-testing framework
<code>jpeg.hpp</code>	Code for reading and writing JPEG files
<code>dog.ppm</code> , <code>crabster.ppm</code> , <code>horses.ppm</code>	Sample input image files.
Several <code>_correct.txt</code> files. Several <code>.correct.ppm</code> files.	Sample (correct) output files used by the <code>processing_public_tests</code> program.
<code>Makefile</code>	Helper commands for building and submitting

Version control

Set up version control using the [Version control tutorial](#).

After you're done, you should have a local repository with a "clean" status and your local repository should be connected to a remote GitHub repository.

```

1  $ git status
2  On branch main
3  Your branch is up-to-date with 'origin/main'.
4
5  nothing to commit, working tree clean
6  $ git remote -v
7  origin  https://github.com/awdeorio/p2-image-processing.git (fetch)
8  origin  https://githubcom/awdeorio/p2-image-processing.git (push)
```

You should have a `.gitignore` file ([instructions](#)).

```
1 $ pwd
2 /Users/awdeorio/src/eecs280/p2-image-processing
3 $ head .gitignore
4 # This is a sample .gitignore file that's useful for C++ projects.
5 ...
```

Group registration

Register your partnership (or working alone) on the Autograder using the direct link in the [Submission and Grading section](#). Then, submit the code you have.

Matrix Module

Create a Matrix abstract data type (ADT). Write implementations in `Matrix.cpp` for the functions declared in `Matrix.hpp`.

Run the public Matrix tests.

```
1 $ make Matrix_public_tests.exe
2 $ ./Matrix_public_tests.exe
```

Complete the EECS 280 [Unit Test Framework Tutorial](#).

Write tests for `Matrix` in `Matrix_tests.cpp` using the unit test framework. You'll submit these tests to the autograder. See the [Unit Test Grading section](#).

```
1 $ make Matrix_tests.exe
2 $ ./Matrix_tests.exe
```

Submit `Matrix.cpp` and `Matrix_tests.cpp` to the Autograder using the link in the [Submission and Grading section](#).

Setup

Rename these files ([VS Code \(macOS\)](#), [VS Code \(Windows\)](#), [Visual Studio](#), [Xcode](#), [CLI](#)):

- `Matrix.cpp.starter` -> `Matrix.cpp`
- `Matrix_tests.cpp.starter` -> `Matrix_tests.cpp`

The Matrix tests should compile and run. Expect them to fail at this point because the `Matrix.cpp` starter code contains function stubs.

```
1 $ make Matrix_public_tests.exe
2 $ ./Matrix_public_tests.exe
3 $ make Matrix_tests.exe
4 $ ./Matrix_tests.exe
```

Configure your IDE to debug either the public tests or your own tests.

	Public tests	Your own tests
VS Code (macOS)	Set program name to: <code>\${workspaceFolder}/Matrix_public_tests.exe</code>	Set program name to: <code>\${workspaceFolder}/Matrix</code>
VS Code (Windows)	Set program name to: <code>\${workspaceFolder}/Matrix_public_tests.exe</code>	Set program name to: <code>\${workspaceFolder}/Matrix</code>
XCode	Include compile sources : <code>Matrix_public_test.cpp</code> , <code>Matrix.cpp</code> , <code>Matrix_test_helpers.cpp</code>	Include compile sources : <code>Matrix_tests.cpp</code> , <code>Matrix</code> <code>Matrix_test_helpers.cpp</code>
Visual Studio	Exclude files from the build: <ul style="list-style-type: none">• Include <code>Matrix_public_test.cpp</code>• Exclude <code>Matrix_tests.cpp</code> , <code>Image_public_test.cpp</code> , <code>Image_tests.cpp</code> , <code>processing_public_tests.cpp</code> , <code>resize.cpp</code> (if present), <code>main.cpp</code> (if present)	Exclude files from the build: <ul style="list-style-type: none">• Include <code>Matrix_tests.c</code>• Exclude <code>Matrix_public</code> <code>Image_public_test.cpp</code> <code>Image_tests.cpp</code> , <code>processing_public_tes</code> <code>resize.cpp</code> (if present), (if present)

Interface

A matrix is a two-dimensional grid of elements. For this project, matrices store integer elements and we will refer to locations by row/column. For example, here's a matrix with 3 rows and 5 columns.

The `Matrix.hpp` file defines a `Matrix` struct to represent matrices and specifies the interface for functions to operate on them. Dimensions of 0 are not allowed.

To create a `Matrix` , first declare a variable and then use an initializer function.

```

1  Matrix m; // create a Matrix object in local memory
2  Matrix_init(&m, 100, 100); // initialize it as a 100x100 matrix

```

Once a `Matrix` is initialized, it is considered valid. Now we can use any of the functions declared in `Matrix.hpp` to operate on it.

```

1  Matrix_fill(&m, 0); // fill with zeros
2
3  // fill first row with ones
4  for (int c = 0; c < Matrix_width(m); ++c) {
5      *Matrix_at(&m, 0, c) = 1; // see description below
6  }
7
8  Matrix_print(&m, cout); // print matrix to cout

```

Access to individual elements in a `Matrix` is provided through a pointer to their location, which can be retrieved through a call to `Matrix_at`. To read or write the element, you just dereference the pointer.

The RMEs in `Matrix.hpp` give a full specification of the interface for each `Matrix` function.

Implementation

The `Matrix` struct looks like this:

```

1  struct Matrix {
2      int width;
3      int height;
4      std::vector<int> data;
5  };

```

`Matrix` stores a 2D grid of numbers in an underlying one-dimensional vector, which in turn stores its elements in a one-dimensional array.

Interface	Implementation
.	.

i Pro-tip: You will need to either create a vector with a given size, or resize an existing vector to that size. Here are some ways to do so:

```

1  // create a vector with 100 elements, all initialized to the value 0
2  std::vector<int> vec(100, 0);
3

```

```

4 // modify an existing vector to have 200 elements with value 0
5 vec.assign(200, 0);
6
7 // replace an existing vector with a new one containing 50 elements,
8 // all initialized to the value 0
9 vec = std::vector<int>(50, 0);

```

Each of the functions in the `Matrix` module takes a pointer to the `Matrix` that is supposed to be operated on. In your implementations of these functions, you should access the `width`, `height`, and `data` members of that `Matrix`, but this is the only place you may do so. To all other code, the individual members are an implementation detail that should be hidden behind the provided interfaces for the `Matrix` module.

Your `Matrix_at` functions will need to perform the appropriate arithmetic to convert from a (row,column) pair to an index in the vector. *This function does not require a loop, and you'll find your implementation will be very slow if you use a loop.*

There are two versions of the `Matrix_at` function to support element access for both const and non-const matrices. The constness of the pointer returned corresponds to the `Matrix` passed in. The implementations for these will be identical.

Remember that you may call any of the functions in a module as part of the implementation of another, and in fact you should do this if it reduces code duplication. In particular, you can access the `data` member directly in the `Matrix_init`, `Matrix_at`, and `Matrix_fill` functions. However, other functions including `Matrix_fill_border`, `Matrix_min_value_in_row`, and `Matrix_column_of_min_value_in_row` will be *easier* to write if they access elements by calling `Matrix_at()` as a helper function instead.

Pro-tip: Use `assert()` to check the conditions in the REQUIRES clause. If other code breaks the interface, that's a bug and you want to know right away! Here's an example.

```

1 // REQUIRES: mat points to a valid Matrix
2 //           0 <= row && row < Matrix_height(&mat)
3 //           0 <= column && column < Matrix_width(&mat)
4 // EFFECTS: Returns a pointer to the element in
5 //           the Matrix at the given row and column.
6 int* Matrix_at(Matrix* mat, int row, int column) {
7     assert(0 <= row && row < mat->height);
8     assert(0 <= column && column < mat->width);
9     // ...
10 }

```

Some things can't be checked, for example that a pointer points to a valid `Matrix` .

Testing

Test your Matrix functions to ensure that your implementations conform to specification in the RME.

Heed the Small Scope Hypothesis. There is no need for large `Matrix` structs. (Other than an as edge case for max size.) Think about what makes tests meaningfully different.

Respect the interfaces for the modules you are testing. Do not access member variables of the structs directly. Do not test inputs that break the REQUIRES clause for a function.

```
1  TEST(test_bad) {
2      Matrix mat;
3      const int width = 3;
4      const int height = 5;
5      const int value = 42;
6      Matrix_init(&mat, 3, 5);
7      Matrix_fill(&mat, value);
8
9      for(int r = 0; r < height; ++r) {
10         for(int c = 0; c < width; ++c) {
11             ASSERT_EQUAL(mat.data[r][c], value); // BAD! DO NOT access member
12             directly
13         }
14     }
```

Sometimes you need to use one Matrix one function while testing another. For example, you need `Matrix_at` to test `Matrix_fill` .

```
1  TEST(test_fill_basic) {
2      Matrix mat;
3      const int width = 3;
4      const int height = 5;
5      const int value = 42;
6      Matrix_init(&mat, 3, 5);
7      Matrix_fill(&mat, value);
8
9      for(int r = 0; r < height; ++r) {
10         for(int c = 0; c < width; ++c) {
11             ASSERT_EQUAL(*Matrix_at(&mat, r, c), value);
12         }
13     }
```



```
14 }
```

Use an `ostringstream` to test `Matrix_print()`.

```
1  #include <sstream>
2
3  TEST(test_matrix_print) {
4      Matrix mat;
5      Matrix_init(&mat, 1, 1);
6
7      *Matrix_at(&mat, 0, 0) = 42;
8      ostringstream expected;
9      expected << "1 1\n";
10     expected << "42 \n";
11     ostringstream actual;
12     Matrix_print(&mat, actual);
13     ASSERT_EQUAL(expected.str(), actual.str());
14 }
```

In your `Matrix` tests, you may use the functions provided in `Matrix_test_helpers.hpp`. Do not use `Image_test_helpers.hpp` in your `Matrix` tests.

Image Module

Create an Image abstract data type (ADT). Write implementations in `Image.cpp` for the functions declared in `Image.hpp`.

Run the public Image tests.

```
1 $ make Image_public_tests.exe
2 $ ./Image_public_tests.exe
```

Write tests for `Image` in `Image_tests.cpp` using the [Unit Test Framework](#). You'll submit these tests to the autograder. See the [Unit Test Grading](#) section.

```
1 $ make Image_tests.exe
2 $ ./Image_tests.exe
```

Submit `Image.cpp` and `Image_tests.cpp` to the Autograder using the link in the [Submission and Grading section](#).

Setup

Rename these files ([VS Code \(macOS\)](#), [VS Code \(Windows\)](#), [Visual Studio](#), [Xcode](#), [CLI](#)):

- `Image.cpp.starter -> Image.cpp`
- `Image_tests.cpp.starter -> Image_tests.cpp`

The Image tests should compile and run. Expect them to fail at this point because the `Image.cpp` starter code contains function stubs.

```
1 $ make Image_public_tests.exe
2 $ ./Image_public_tests.exe
3 $ make Image_tests.exe
4 $ ./Image_tests.exe
```

Write tests for `Image` in `Image_tests.cpp` using the [Unit Test Framework](#). You'll submit these tests to the autograder. See the [Unit Test Grading](#) section.

```
1 $ make Image_tests.exe
2 $ ./Image_tests.exe
```










Configure your IDE to debug either the public tests or your own tests.

	Public tests	Your own tests
VS Code (macOS)	Set program name to: <code>\${workspaceFolder}/Image_public_tests.exe</code>	Set program name to: <code>\${workspaceFolder}/Image_t</code>
VS Code (Windows)	Set program name to: <code>\${workspaceFolder}/Image_public_tests.exe</code>	Set program name to: <code>\${workspaceFolder}/Image_t</code>
XCode	Include compile sources : <code>Image_public_test.cpp</code> , <code>Matrix.cpp</code> , <code>Image.cpp</code> , <code>Matrix_test_helpers.cpp</code> , <code>Image_test_helpers.cpp</code>	Include compile sources : <code>Image_tests.cpp</code> , <code>Matrix.c</code> <code>Image.cpp</code> , <code>Matrix_test_helpers.cpp</code> , <code>Image_test_helpers.cpp</code>

	Public tests	Your own tests
Visual Studio	<div>Exclude files from the build :</div> <ul style="list-style-type: none">• Include <code>Image_public_test.cpp</code>• Exclude <code>Image_tests.cpp</code> , <code>Matrix_public_test.cpp</code> , <code>Matrix_tests.cpp</code> , <code>processing_public_tests.cpp</code> , <code>resize.cpp</code> (if present), <code>main.cpp</code> (if present)	<div>Exclude files from the build:</div> <ul style="list-style-type: none">• Include <code>Image_tests.cpp</code>• Exclude <code>Image_public_test.cpp</code> , <code>Matrix_public_test.cpp</code> , <code>Matrix_tests.cpp</code> , <code>processing_public_test</code> <code>resize.cpp</code> (if present), <code>main.cpp</code> (if present)

Interface

An `Image` is similar to a `Matrix` , but contains `Pixel` s instead of integers. Each `Pixel` includes three integers, which represent red, green, and blue (RGB) color components. Each component takes on an intensity value between 0 and 255. The `Pixel` type is considered “Plain Old Data” (POD), which means it doesn’t have a separate interface. We just use its member variables directly. Here is the `Pixel` struct and some examples:

<pre>struct Pixel { int r; // red int g; // green int b; // blue }</pre>	 (255,0,0)	 (0,255,0)	 (0,0,255)
	 (0,0,0)	 (255,255,255)	 (100,100,100)
	 (101,151,183)	 (124,63,63)	 (163,73,164)

Below is a 5x5 image and its conceptual representation as a grid of pixels.

.

Dimensions of 0 are not allowed. To create an `Image` , first declare a variable and then use an initializer function. There are several initializer functions, but for now we’ll just use the basic one.

```
1 Image img; // create an Image object in local memory  
2 Image_init(&img, 5, 5); // initialize it as a 5x5 image
```

Once an `Image` is initialized, it is considered valid. Now we can use any of the functions declared in `Image.hpp` to operate on it.


```
1 Pixel um_blue = { 0, 46, 98 };
2 Pixel um_maize = { 251, 206, 51 };
3 Image_fill(&img, um_blue); // fill with blue
4
5 // fill every other column with maize to make stripes
6 for (int c = 0; c < Image_width(&img); ++c) {
7     if (c % 2 == 0) { // only even columns
8         for (int r = 0; r < Image_height(&img); ++r) {
9             Image_set_pixel(&img, r, c, um_maize);
10        }
11    }
12 }
```

To read and write individual `Pixel`s in an `Image`, use the `Image_get_pixel` and `Image_set_pixel` functions, respectively.

The RMEs in `Image.hpp` give a full specification of the interface for each `Image` function.

PPM Format

The `Image` module also provides functions to read and write `Image`s from/to the PPM image format. Here’s an example of an `Image` and its representation in PPM.

Image	Image Representation in PPM
	<pre>P3 5 5 255 0 0 0 0 0 0 255 255 250 0 0 0 0 0 0 255 255 250 126 66 0 126 66 0 126 66 0 255 255 250 126 66 0 0 0 0 255 219 183 0 0 0 126 66 0 255 219 183 255 219 183 0 0 0 255 219 183 255 219 183 255 219 183 0 0 0 134 0 0 0 0 255 219 183</pre>

The PPM format begins with these elements, each separated by whitespace:

- `p3` (Indicates it’s a “Plain PPM file”.)
- `WIDTH HEIGHT` (Image width and height, separated by whitespace.)
- `255` (Max value for RGB intensities. We’ll always use 255.)

This is followed by the pixels in the image, listed with each row on a separate line. A pixel is written as three integers for its RGB components in that order, separated by whitespace.

To write an image to PPM format, use the `Image_print` function that takes in a `std::ostream`. This can be used in conjunction with file I/O to write an image to a PPM file. **The `Image_print` function must produce a PPM using whitespace in a very specific way** so that we can use `diff` to compare your output PPM file against a correct PPM file. See the RME for the full details.

To create an image by reading from PPM format, use the `Image_init` function that takes in a `std::istream`. This can be used in conjunction with file I/O to read an image from a PPM file. Because we may be reading in images generated from programs that don't use whitespace in the same way that we do, the `Image_init` function must accommodate any kind of whitespace used to separate elements of the PPM format (if you use C++ style I/O with `>>`, this should be no problem). Other than variance in whitespace (not all PPM files put each row on its own line, for example), you may assume any input to this function is in valid PPM format. (Some PPM files may contain "comments", but you do not have to account for these.)

See [Working with PPM Files](#) for more information on working with PPM files and programs that can be used to view or create them on various platforms.

Implementation

The `Image` struct looks like this:

```
1  struct Image {
2      int width;
3      int height;
4      Matrix red_channel;
5      Matrix green_channel;
6      Matrix blue_channel;
7  };
```

The Interface for `Image` makes it seem like we have a grid of `Pixel`s, but the `Image` struct actually stores the information for the image in three separate `Matrix` structs, one for each of the RGB color channels. There are no `Pixel`s in the underlying representation, so your `Image_get_pixel` function must pack the RGB values from each color `Matrix` into a `Pixel` to be returned. Likewise, `Image_set_pixel` must unpack RGB values from an input `Pixel` and store them into each `Matrix`.

Each of the functions in the `Image` module takes a pointer to the `Image` that is supposed to be operated on. When you are writing implementations for these functions, you may be tempted to access members of the `Matrix` struct directly (e.g. `img->red_channel.width`, `img->green_channel.data[x]`). Don't do it! They aren't part of the interface for `Matrix`, and you should

not use them from the outside. Instead, use the `Matrix` functions that are part of the interface (e.g. `Matrix_width(&img->red_channel)` , `Matrix_at(&img->green_channel, r, c)`).

In your implementation of the `Image_init` functions, space for the `Matrix` members will have already been allocated as part of the `Image` . However, you still need to initialize these with a call to `Matrix_init` to ensure they are the right size!

The `Image` struct contains `width` and `height` members. These are technically redundant, since each of the `Matrix` members also keeps track of a width and height, but having them around should make the implementations for your functions easier to read.

Respect the Interfaces!

Our goal is to use several modules that work together through well-defined interfaces, and to keep the implementations separate from those interfaces. The interfaces consist of the functions we provide in the `.hpp` file for the module, but NOT the member variables of the struct. The member variables are part of the implementation, not the interface!

This means you may access member variables directly (i.e. using `.` or `->`) when you're writing code within their module, but never from the outside world! For example, let's consider the `Image` module. If I'm writing the implementation of a function inside the module, like `Image_print` , it's fine to use the member variable `height` directly:

```
1 void Image_print(const Image *img, std::ostream& os) {
2     ...
3     // loop through all the rows
4     for (int r = 0; r < img->height; ++r) {
5         // do something
6     }
7     ...
8 }
```

This is fine, because we assume the person who implements the module is fully aware of all the details of how to use `height` correctly. However, if I'm working from the outside, then using member variables directly is very dangerous. This code won't work right:

```
1 int main() {
2     // Make a 400x300 image (sort of)
3     Image img;
4     img.width = 400;
5     img.height = 300;
6     // do something with img but it doesn't work :(
7     ...
8 }
```

The problem is that we “forgot” about initializing the width and height of the `Matrix` structs that make up each color channel in the image. Instead, we should have used the `Image_init` function from the outside, which takes care of everything for us.

Here’s the big idea - we don’t want the “outside world” to have to worry about the details of the implementation, or even to know them at all. We want to support substitutability, so that the implementation can change without breaking outside code (as long as it still conforms to the interface). Using member variables directly from the outside messes this all up. Don’t do it! It could break your code and this will be tested on the autograder!

An exception to this rule is the `Pixel` struct. It’s considered to be a “Plain Old Data” (POD) type. In this case, the interface and the implementation are the same thing. It’s just an aggregate of three ints to represent an RGB pixel - nothing more, nothing less.

We should note there are patterns used in C-style programming that hide away the definition of a struct’s members and prevent us from accidentally accessing them outside the correct module. Unfortunately, this causes complications that we don’t have all the tools to deal with yet (namely dynamic memory management). We’ll also see that C++ adds some built-in language mechanisms to control member accessibility. For now, you’ll just have to be careful!

Copying Large Structs

In many cases you will find it useful to copy `Matrix` and `Image` structs in your code. This is supported by the interface, so feel free to use it wherever useful. However, try to avoid making unnecessary copies, as this can slow down your code.

As an example, let’s say you wanted to add a border to a `Matrix` and print it without changing the original. You could write this:

```
1  ...
2  // Assume we have a variable mat that points to a Matrix
3
4  // Make a copy of mat and add the border. original remains unchanged
5  Matrix mat_border = *mat; // need to dereference mat to copy it
6  Matrix_fill_border(&mat_border, 0);
7
8  // print the bordered version
9  Matrix_print(&mat_border, os);
10 ...
```

Testing

Respect the interfaces for the modules you are testing. Do not access member variables of the structs directly. Do not test inputs that break the `REQUIRES` clause for a function.

You may use stringstream to simulate file input and/or output for your unit tests. You may also use the image files `dog.ppm`, `crabster.ppm`, and `horses.ppm`, but no others.

In your `Image` tests, you may use the functions provided in `Image_test_helpers.hpp`. Do not use `Matrix_test_helpers.hpp` in your `Image` tests.

Processing Module

The `processing` module contains several functions that perform image processing operations. Some of these provide an interface for content-aware resizing of images, while others correspond to individual steps in the seam carving algorithm.

The main interface for using content-aware resizing is through the `seam_carve`, `seam_carve_width` and `seam_carve_height` functions. These functions use the seam carving algorithm to shrink either an image's width or height in a context-aware fashion. The `seam_carve` function adjusts both width and height, but width is always done first. For this project, we only support shrinking an image, so the requested width and height will always be less than or equal to the original values.

Write implementations in `processing.cpp` for the functions declared in `processing.hpp`.

Run the public processing tests.

```
1 $ make processing_public_tests.exe
2 $ ./processing_public_tests.exe
```

Submit `processing.cpp` to the Autograder using the link in the [Submission and Grading section](#).

Setup

Rename this file ([VS Code \(macOS\)](#), [VS Code \(Windows\)](#), [Visual Studio](#), [Xcode](#), [CLI](#)):

- `processing.cpp.starter` -> `processing.cpp`

The Processing tests should compile and run. Expect them to fail at this point because the `processing.cpp` starter code contains function stubs.

```
1 $ make processing_public_tests.exe
2 $ ./processing_public_tests.exe
```

Configure your IDE to debug public tests.

VS Code (macOS)	Set program name to: <code>\${workspaceFolder}/processing_public_tests.exe</code>
VS Code (Windows)	Set program name to: <code>\${workspaceFolder}/processing_public_tests.exe</code>
XCode	Include compile sources : <code>processing_public_tests.cpp</code> , <code>Matrix.cpp</code> , <code>Image.cpp</code> , <code>processing.cpp</code> , <code>Matrix_test_helpers.cpp</code> , <code>Image_test_helpers.cpp</code>
Visual Studio	Exclude files from the build: <ul style="list-style-type: none">• Include <code>processing_public_tests.cpp</code>• Exclude <code>Matrix_public_test.cpp</code> , <code>Matrix_tests.cpp</code> , <code>Image_public_test.cpp</code> , <code>Image_tests.cpp</code> , <code>resize.cpp</code> (if present), <code>main.cpp</code> (if present)

Energy Matrix

`compute_energy_matrix` : The seam carving algorithm works by removing seams that pass through the least important pixels in an image. We use a pixel's energy as a measure of its importance.

To compute a pixel's energy, we look at its neighbors. We'll call them N (north), S (south), E (east), and W (west) based on their direction from the pixel in question (we'll call it X).

The energy of X is the sum of the squared differences between its N/S and E/W neighbors:

$$\text{energy}(X) = \text{squared_difference}(N, S) + \text{squared_difference}(W, E)$$

The static function `squared_difference` is provided as part of the starter code. Do not change the implementation of the `squared_difference` function.

To construct the energy `Matrix` for the whole image, your function should do the following:

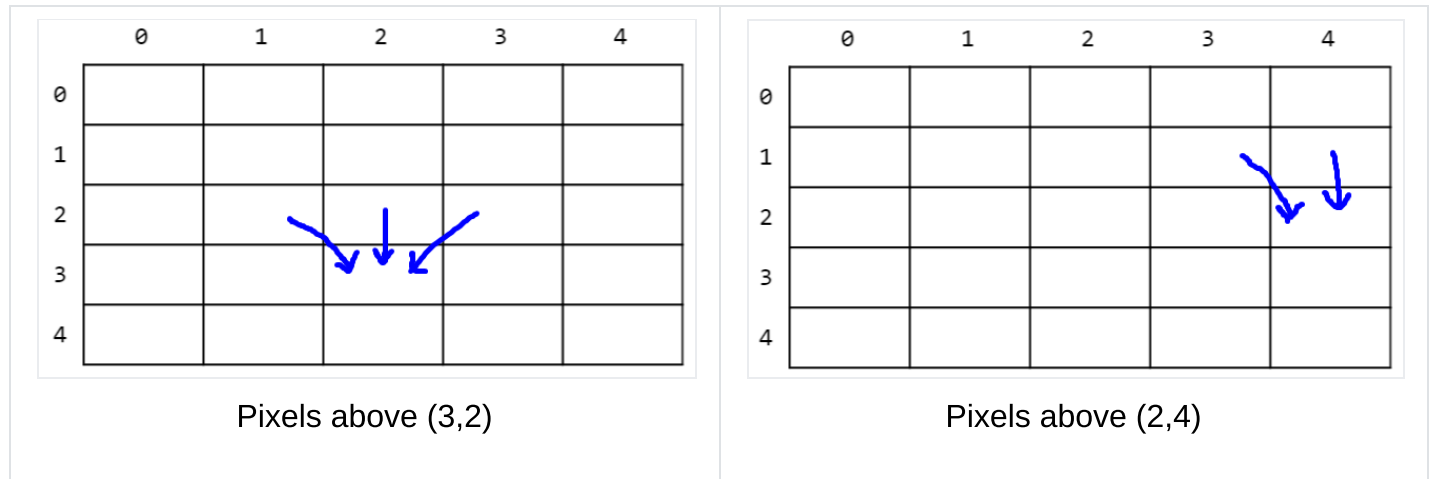
1. Initialize the energy `Matrix` with the same size as the `Image` and fill it with zeros.
2. Compute the energy for each non-border pixel, using the formula above.
3. Find the maximum energy so far, and use it to fill in the border pixels.

Cost Matrix

`compute_vertical_cost_matrix` : Once the energy matrix has been computed, the next step is to find the path from top to bottom (i.e. a vertical seam) that passes through the pixels with the lowest total energy (this is the seam that we would like to remove).

We will begin by answering a related question - given a particular pixel, what is the minimum energy we must move through to get to that pixel via any possible path? We will refer to this as the cost of that pixel. Our goal for this stage of the algorithm will be to compute a matrix whose entries correspond to the cost of each pixel in the image.

Now, to get to any pixel we have to come from one of the three pixels above it.



We would want to choose the least costly from those pixels, which means the minimum cost to get to a pixel is its own energy plus the minimum cost for any pixel above it. This is a recurrence relation. For a pixel with row `r` and column `c`, the cost is:

```
1  cost(r, c) = energy(r, c) + min(cost(r-1, c-1),
2                                   cost(r-1, c),
3                                   cost(r-1, c+1))
```

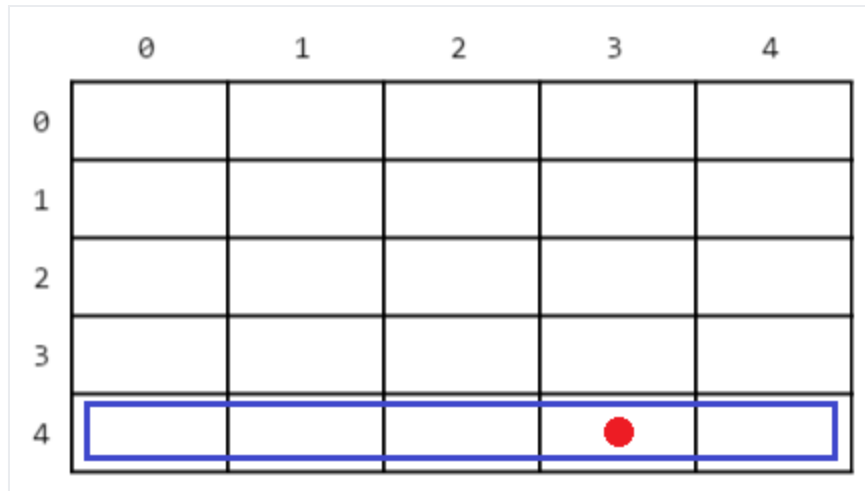
Use the `Matrix_min_value_in_row` function to help with this equation. Of course, you need to be careful not to consider coming from pixels outside the bounds of the `Matrix`.

We could compute costs recursively, with pixels in the first row as our base case, but this would involve a lot of repeated work since our subproblems will end up overlapping. Instead, let's take the opposite approach...

1. Initialize the cost `Matrix` with the same size as the energy `Matrix`.
2. Fill in costs for the first row (index 0). The cost for these pixels is just the energy.
3. Loop through the rest of the pixels in the `Matrix`, row by row, starting with the second row (index 1). Use the recurrence above to compute each cost. Because a pixel's cost only depends on other costs in an earlier row, they will have already been computed and can just be looked up in the `Matrix`.

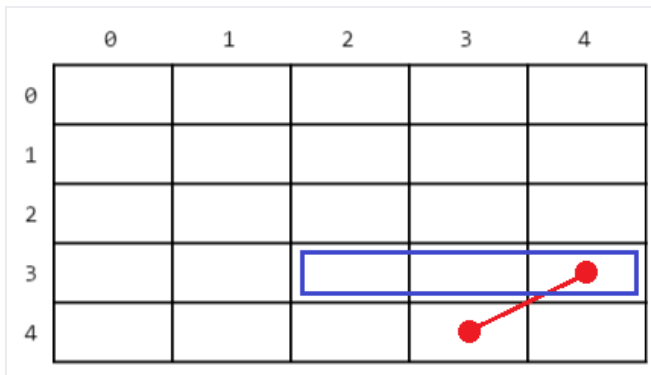
Minimal Vertical Seam

`find_minimal_vertical_seam` : The pixels in the bottom row of the image correspond to the possible endpoints for any seam, so we start with the one of those that is lowest in the cost matrix.



First, find the minimum cost pixel in the bottom row.

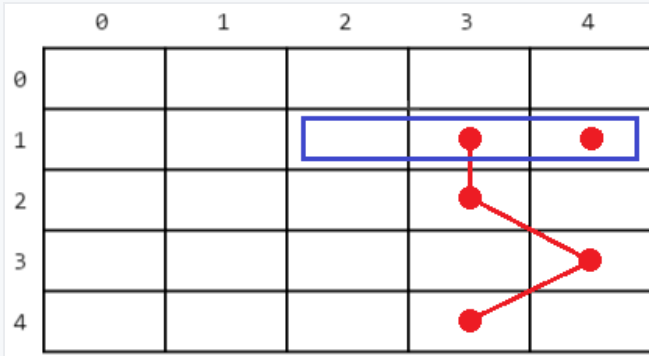
Now, we work our way up, considering where we would have come from in the row above. In the pictures below, the blue box represents the “pixels above” in each step.



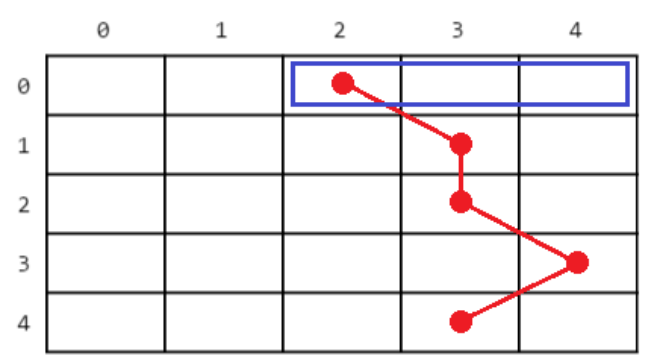
Then find the minimum cost pixel above.



Don't look outside the bounds!



For ties, pick the leftmost.

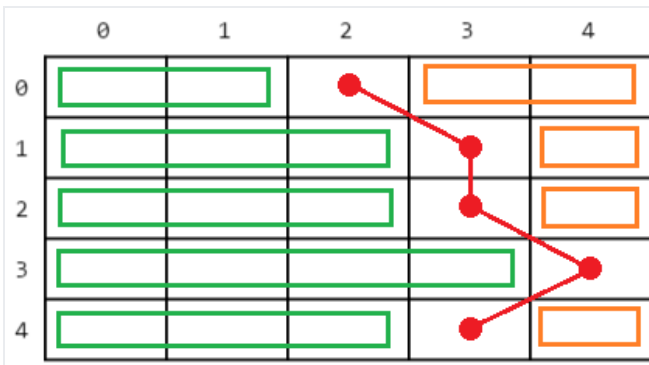


Repeat until you reach the top row.

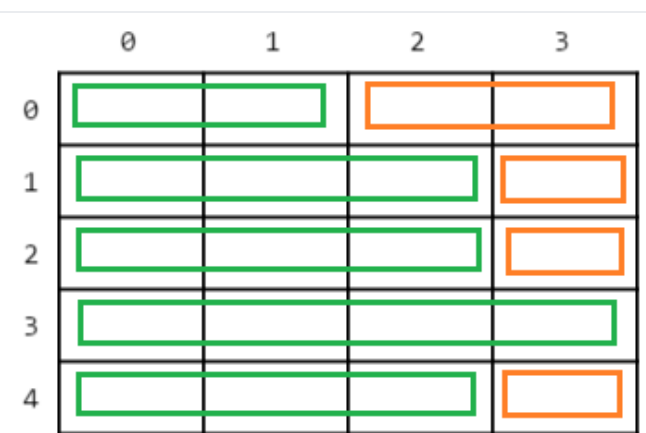
You will find the `Matrix_column_of_min_value_in_row` function useful here. Each time you process a row, put the column number of the best pixel in the seam vector, working your way from the back to front. (i.e. The last element corresponds to the bottom row.)

Removing a Vertical Seam

`remove_vertical_seam` : The seam vector passed into this function contains the column numbers of the pixels that should be removed in each row, in order from the top to bottom rows. To remove the seam, copy the image one row at a time, first copying the part of the row before the seam (green), skipping that pixel, and then copying the rest (orange).



Original



Seam Removed

You should copy into a smaller auxiliary `Image` and then back into the original, because there is no way to change the width of an existing image. Do not attempt to use `Image_init` to “resize” the original - it doesn’t preserve existing data in an `Image`.

Seam Carving Algorithm

We can apply seam carving to the width of an image, the height, or both.

`seam_carve_width`

To apply seam carving to the width, remove the minimal cost seam until the image has reached the appropriate width.

1. Compute the energy matrix
2. Compute the cost matrix
3. Find the minimal cost seam
4. Remove the minimal cost seam

`seam_carve_height`

To apply seam carving to the height, just do the following:

1. Rotate the image left by 90 degrees
2. Apply `seam_carve_width`
3. Rotate the image right by 90 degrees

`seam_carve`

To adjust both dimensions:

1. Apply `seam_carve_width`
2. Apply `seam_carve_height`

Testing

We have provided the `processing_public_tests.cpp` file that contains a test suite for the seam carving algorithm that runs each of the functions in the `processing` module and compares the output to the “`_correct`” files included with the project.

You should write your own tests for the `processing` module, but you do not need to turn them in. You may do this either by creating a copy of `processing_public_tests.cpp` and building onto it, or writing more tests from scratch. Pay attention to edge cases.

Use the Makefile to compile the test with this command:

```
$ make processing_public_tests.exe
```

Then you can run the tests for the `dog` , `crabster` , and `horses` images as follows:

```
$ ./processing_public_tests.exe
```

You can also run the tests on just a single image:

```
1 $ ./processing_public_tests.exe test1_dog
2 $ ./processing_public_tests.exe test2_crabster
3 $ ./processing_public_tests.exe test3_horses
```

When the test program runs, it will also write out image files containing the results from your functions before asserting that they are correct. You may find it useful to look at the results from your own code and visually compare them to the provided correct outputs when debugging the algorithm.

The seam carving tests work sequentially and stop at the first deviation from correct behavior so that you can identify the point at which your code is incorrect.

Resize Program

The main resize program supports content-aware resizing of images via a command line interface.

Create a `resize.cpp` file and write your implementations of the driver program there.

Compile and run the program. The [interface](#) section explains each command line argument.

```
$ make resize.exe
$ ./resize.exe horses.ppm horses_400x250.ppm 400 250
```

Setup

If you created a `main.cpp` while following the setup tutorial, rename it to `resize.cpp` . Otherwise, create a new file `resize.cpp` ([VS Code \(macOS\)](#), [VS Code \(Windows\)](#), [Visual Studio](#), [Xcode](#), [CLI](#)).

Add “hello world” code if you haven’t already.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout << "Hello World!\n";
6 }
```

The resize program should compile and run.

```
1 $ make resize.exe
```

```
2 $ ./resize.exe
3 Hello World!
```

Configure your IDE to debug the resize program.

VS Code (macOS)	Set program name to: \${workspaceFolder}/resize.exe
VS Code (Windows)	Set program name to: \${workspaceFolder}/resize.exe
XCode	Include compile sources : resize.cpp , Matrix.cpp , Image.cpp , processing.cpp
Visual Studio	Exclude files from the build: <ul style="list-style-type: none">• Include resize.cpp• Exclude Matrix_public_test.cpp , Matrix_tests.cpp , Image_public_test.cpp , Image_tests.cpp , processing_public_tests.cpp , main.cpp (if present).

Configure command line arguments ([VS Code \(macOS\)](#), [VS Code \(Windows\)](#), [Xcode](#), [Visual Studio](#)). We recommend starting with the smallest input, `dog.ppm dog_4x5.out.ppm 4 5`.

To compile, run, and test the smallest input at the command line:

```
1 $ make resize.exe
2 $ ./resize.exe dog.ppm dog_4x5.out.ppm 4 5
3 $ diff dog_4x5.out.ppm dog_4x5.correct.ppm
```

Interface

To resize the file `horses.ppm` to be 400x250 pixels and store the result in the file `horses_400x250.ppm`, we would use the following command:

```
$ ./resize.exe horses.ppm horses_400x250.ppm 400 250
```

In particular, here’s what each of those means:

Argument	Meaning
<code>horses.ppm</code>	The name of the input file from which the image is read.

Argument	Meaning
<code>horses_400x250.ppm</code>	The name of the output file to which the image is written.
<code>400</code>	The desired width for the output image.
<code>250</code>	The desired height for the output image. (Optional)

The program is invoked with three or four arguments. If no height argument is supplied, the original height is kept (i.e. only the width is resized). If your program takes about 30 seconds for large images, that's ok. There's a lot of computation involved.

Error Checking

The program checks that the command line arguments obey the following rules:

- There are 4 or 5 arguments, including the executable name itself (i.e. `argv[0]`).
- The desired width is greater than 0 and less than or equal to the original width of the input image.
- The desired height is greater than 0 and less than or equal to the original height of the input image.

If any of these are violated, use the following lines of code (literally) to print an error message.

```
1  cout << "Usage: resize.exe IN_FILENAME OUT_FILENAME WIDTH [HEIGHT]\n"  
2      << "WIDTH and HEIGHT must be less than or equal to original" << endl;
```

Your program should then exit with a non-zero return value from `main` . Do **not** use the `exit` function in the standard library, as it does not clean up local objects.

If the input or output files cannot be opened, use the following lines of code (literally, except change the variable `filename` to whatever variable you have containing the name of the problematic file) to print an error message, and then return a non-zero value from `main` .

```
cout << "Error opening file: " << filename << endl;
```

i You do not need to do any error checking for command line arguments or file I/O other than what is described in this section. However, you must use precisely the error messages given here in order to receive credit.

Implementation

Your `main` function should not contain much code. It should just process the command line arguments, check for errors, and then call the appropriate functions from the other modules to perform the desired task.

Optional: Reading and Writing JPEG Files

Your program must handle input and output files in the PPM format, but you may optionally also implement support for files in the JPEG format. To do so, follow these steps:

1. Install the `libjpeg` or `libjpeg-turbo` library. On MacOS, you can run the following:

```
$ brew install jpeg-turbo
```

On WSL or Linux, run the following:

```
$ sudo apt install libjpeg-turbo8-dev
```

2. Edit the `Makefile` and set `USE_LIBJPEG` to `true`:

```
USE_LIBJPEG ?= true
```

On MacOS, you will also need to set `LIBJPEG_PATH`. Run the following in the terminal:

```
$ brew info jpeg-turbo
```

You should see output like the following:

```
==> jpeg-turbo: stable 3.0.2 (bottled), HEAD
JPEG image codec that aids compression and decompression
https://www.libjpeg-turbo.org/
/opt/homebrew/Cellar/jpeg-turbo/3.0.2 (44 files, 3.4MB) *
...
```

Set `LIBJPEG_PATH` to the path you see in the output, e.g.

```
LIBJPEG_PATH ?= /opt/homebrew/Cellar/jpeg-turbo/3.0.2
```

3. In `resize.cpp`, add the following `#include`. Do not add it in any other files.

```
#include "jpeg.hpp"
```

You can then make use of the following functions defined in `jpeg.hpp`:

- `has_jpeg_extension()` determines whether a file name ends with `.jpg` or `.jpeg`, ignoring capitalization – use this to determine whether the files specified at the command line are JPEG files

- `read_jpeg()` reads a JPEG image from a file into an `Image` object
- `write_jpeg()` writes an image from an `Image` object into a JPEG file

See the full documentation of each function in `jpeg.hpp`.

4. Once you are sure that your program is working, you may wish to compile with optimization enabled to make it run significantly faster:

```
1 $ make clean
2 $ make resize.exe CXXFLAGS="--std=c++17 -O3"
```

Even with optimizations enabled, seam carving is quite expensive. We recommend using input images that are no larger than 1000x1000 pixels.

Submission and Grading

Submit to the autograder using this direct autograder link: <https://autograder.io/web/project/3149>.

- `Matrix.cpp`
- `Matrix_tests.cpp`
- `Image.cpp`
- `Image_tests.cpp`
- `processing.cpp`
- `resize.cpp`

This project will be autograded for correctness, comprehensiveness of your test cases, and programming style. See the [style checking tutorial](#) for the criteria and how to check your style automatically on CAEN.

Testing

Run all the unit tests and system tests. This includes the public tests we provided and the unit tests that you wrote.

```
$ make test
```

i Pro-tip: Run commands in parallel with `make -j`.

```
$ make -j4 test
```

Unit Test Grading

We will autograde your `Matrix` and `Image` unit tests.

Your unit tests must use the [unit test framework](#).

A test suite must complete less than 5 seconds and contain 50 or fewer `TEST()` items. One test suite is one `_test.cpp` file.

To grade your unit tests, we use a set of intentionally buggy instructor solutions. You get points for catching the bugs.

1. We compile and run your unit tests with a **correct solution**.
 - Tests that pass are **valid**.
 - Tests that fail are **invalid**, they falsely report a bug.
2. We compile and run all of your **valid** tests against each **buggy solution**.
 - If any of your tests fail, you caught the bug.
 - You earn points for each bug that you catch.

Requirements and Restrictions

It is our goal for you to gain practice with good C-style object-based programming and proper use of pointers and structs. Here are some (mandatory) guidelines.

DO	DO NOT
Use <i>either</i> traversal by pointer or traversal by index, as necessary	
Modify <code>.cpp</code> files	Modify <code>.hpp</code> files
Put any extra helper functions in the <code>.cpp</code> files and declare them <code>static</code>	Modify <code>.hpp</code> files
	<code>#include</code> a <code>.hpp</code> file from a module that does not require the code in the <code>.hpp</code> file (e.g. including <code>Image.hpp</code> from <code>Matrix.cpp</code>), as this introduces an incorrect dependency between modules
<code>#include</code> a library to use its functions	Assume that the compiler will find the library for you (some do, some don't)

DO	DO NOT
Pass large structs or classes by pointer	Pass large structs or classes by value
Pass by pointer-to-const when appropriate	“I don’t think I’ll modify it ...”

Diagnosing Slow Code

If your code runs too slowly (especially on larger images like the “horses” example), a tool called `perf` can analyze which parts of your code take the most time. See the [Perf Tutorial](#) for details.

Undefined Behavior

If your code produces different results on different machine (e.g. your computer vs. the autograder), the likely source is undefined behavior. Refer to the [Sanitizers Tutorial](#) for how to use the Address Sanitizer (ASAN) to check for undefined behavior.

Reach Goals

Optionally check out the project [reach goals](#). Reach goals are entirely optional.

Acknowledgments

This project was written by James Juett, Winter 2016 at the University of Michigan. It was inspired by Josh Hug’s “Nifty Assignment” at SIGCSE 2015.