Homework

29

The Image Class

Previously, you used raw pointers, structures and arrays to manipulate digital images. I'm certain, as you worked on those image exercises, that you thought, if only to yourself, **"there has to be a better way!"** Well, you're right, **there is**. Encapsulation and classes give us the ability to hide all of those messy details behind an elegant interface.

Open the implementation (**.cpp**) file in the editor a look at **run()** at the bottom (under student testing), which uses the **Image** class from the user's perspective. (This code was written before a single line of the **Image** class was completed.)

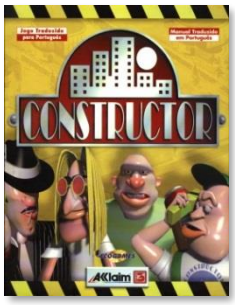
Then, open the header file and look at the interface for the **Image** class; it is entirely complete. Follow along as we use the interface portion of the class.

1. Examine the Interface

Here is the first part. The user can create **Image** objects in three different ways.

```
Image empty;                    // empty image
Image square(20, 20);           // square image
Image lego("images/lego.png");  //  from file
```

1. A default **Image**, like **empty**, can load different images later.
2. An **Image** with a given size, such as ***square*** can have its pixels modified.
3. An **Image** can be initialized using a **jpg**, **bmp** and **png** file on disk.



These requirements are met with these three constructors:

- A **default constructor** as used to create the **Image empty**.
- An **overloaded constructor** that takes a file path.
- **Another overloaded constructor** that takes two **int**s for the size

Image Accessors

Section #2 calls the different **accessor member functions** to determine the state of each **Image** object immediately after creation, passing each object to the **info()** helper function (defined at the very bottom of the file).

```
void info(const string& msg, const Image& img)
{
    cout << msg << ": size()->" << img.size()
        << ", width()->" << img.width()
        << ", height()->" << img.height() << endl;
}
```

Accessors often start with **get**, but in C++ that is often **implied**. Notice the **explicit const** in the header of each of these that that indicates they are accessors. This indicates that this member function is **not permitted** to change the state of the object. The compiler will enforce this restriction.

Image Mutators

Section #3 calls the **two mutators**. The first mutator is load(). We can load a file after the **Image** object has been created, **reading the entire contents into memory as well**.

```
bool ok = empty.load("images/paris.jpg");
string msg = "Loading paris into empty image. ";
msg += ok ? "Success." : "Failed.";
cout << msg << endl;
info("empty", empty);
```

Instead of throwing exceptions, the **Image** acts like more like the C++ streams. ***load()*** returns a Boolean indicating if it succeeded or not.

The second mutator is ***fill()***, which sets all the pixels in the image with the supplied color value. The function accepts a ***Pixel*** object, in the form **RGBA**, for ***red***, ***green***, ***blue***, and ***alpha***. If you don't supply the values, then they default to ***0***, or, in the case of alpha, to **255**, which means non-transparent.

```
Pixel fillColor{255, 127};
square.fill(fillColor);
```

Element Access

To access the individual elements (pixels) in the picture, you can use the **range *for* loop**, or an iterator loop. To support this, **Image** has both a **begin()** and an **end()** method.

begin() returns a pointer to the first pixel, and **end()** returns a pointer to the location after the last pixel. If the **Image** is empty, they both return **nullptr**.

Here's the first part of Section #4 using the **range *for* loop** to grayscale **empty**.

```
for (auto& p : empty) // grayscale empty (contains Paris)
{
    auto avg = (p.red + p.green + p.blue) / 3;
    p.red = p.green = p.blue = avg;
}
```

Here's a fragment of the second part that uses an iterator loop to draw a cross on the **Image** named **square**.

```
for (auto itr = begin(square); itr != end(square); ++itr)
{
    . . .
}
```

Saving the Files

The last prototype is a function to save the file. Here's code from **run()**:

```
auto pOK = empty.save("actual/paris.png");
auto sOK = square.save("actual/square.png");
auto lOK = lego.save("actual/lego.jpg");
```

This mirrors the ***Load()*** function.

2. Write the Stubs

Now that we have the interface developed, **it's time to implement it**. And, the first step, as always, is to **write the skeletons or stubs**. Here are the steps:

1. Copy the member function **prototypes** from the header file to the **.cpp** file. **Do not** copy the prototype that has **=default** at the end.
2. Add the **class scope qualifier**, **Image::**, to the front of each name.
3. Remove the **semicolons and add braces** for each body.
4. Do nothing more for the **void** functions.
5. Have the **int** accessors return **1** (not **0**).
6. Have the **begin()** and **end()** member functions return **nullptr**
7. The others may return **false**

Once you have the stubs, then type **`make test`** to check if the mechanics are complete. If you have **any** errors at this point, don't go on. Check on Piazza for help with those or come into my office hour.

3. Implementing Image

Below is a list of hints that will help you implement the **`Image`** class.

1. The two-argument constructor initializes **`m_width`** and **`m_height`**, as well as **`m_pixels`**. Initialize **`m_pixels`** using the **`vector(int)`** constructor which creates a **`vector`** containing **`width * height Pixels`**.
2. Accessors return **`m_width`**, **`m_height`** and **`m_pixels.size()`** respectively.
3. The **`fill()`** mutator may use a loop or **`m_pixels.assign()`**.
4. **`begin()`** and **`end()`** return the address the first pixel and the address appearing immediately after the last pixel, respectively. But, if the **`vector`** is empty, they both **`nullptr`**.
5. The **`load()`** member function:
    a. Call **`stbi_load()`** to load the image and initialize width, height and bits-per-pixel. **Save the returned value in a temporary pointer**. Look at the appendix at the end if you don't remember how
    b. If the returned value is not null, then:
        i. Assign width and height to the data members
        ii. Use **`m_pixels.resize()`** to allocate enough memory, and copy each pixel into the **`vector`**. It is easiest if you cast the pointer to a **`Pixel*`** as before.
        iii. Pass the returned pointer to **`stbi_image_free()`**.
        iv. Return **`true`**
    c. Else return **`false`**
6. For the file-path constructor, just call **`load()`**.
7. The ***`save`*** function is the most difficult. Here is the basic plan:

```
1. Extract the extension and convert to lower case
2. Write a sequential if statement for each type
   3. Call the appropriate save function
```

    a. Fnd the dot in front of the extension using **`rfind()`** (not **`find()`**). Then, use **`substr()`** to extract the extension.
    b. At any point if you fail, return ***`false`***

c. Use a range-based loop to convert the extension to lowercase. Pass the element by reference, not by value.

d. Because **stbi_save()** is a C function, file names must be C-style. The C++ string member **c_str()** does that.

e. Use the accessors or the data members to get the width & height.

f. Always save 4 bits per pixel. (Add a constant **BPP**)

g. **PNG** and **jpg** files require an additional argument when saving. Use **4 * width** for **PNG** and **100** for **jpg**.

h. To save each of the image types, you must supply the address of the data. Use **&m_pixels.front()** function to get that pointer.

i. Our pixels are stored in a **vector<Pixel>**, but the C-library **save()** function is expecting a pointer to an **unsigned char**. Fortunately, **reinterpret_cast** is just what we need.

j. The code for **BMP** and **JPG** files is similar. For **JPG** files, instead of a "stride" calculation, it requires the quality for the image. Use a constant **100** (highest quality) there. The **BMP** save function only has **5** arguments, not **6**. See the code in the Appendix below.

Run the "example" with **make run**. You'll find the files produced in the **actual** folder.

Run the instructor tests by using **make test**. Submit as usual with **make submit**. If you need help, check on Piazza or come into my office hour.

Appendix: the stb_image Functions

We are using the ***stb_image*** and ***stb_image_write*** libraries, written by Sean T. Barret (the stb) and placed into the public domain. These libraries provide the ability to read and write several different image formats, in several different ways. Both are C libraries instead of C++ libraries.

Loading a JPEG Image

```
const int CHANNELS = 4;
int width, height, bpp;
unsigned char *p = stbi_load("images/paris.jpg",
            &width, &height, &bpp,  // output parameters
            CHANNELS);
```

This code loads an image from your disk into memory.

- ***stbi_load()*** returns a **pointer** to the first byte of the image in memory as an **unsigned char***. If loading fails, it returns **nullptr**.
- The first argument is the **path to the file as a C-style string**.
- The next three arguments are the **addresses of the width, height**, and **bytes-per-pixel** of the image. These are **output parameters**; you create the variables, pass **their addresses** as arguments, and the function will **fill them in**. The information flows **out of the function**, not into it.
- The last argument is an **input** parameter telling ***load()*** how to interpret the image. We'll always use ***4*** channels of input.

Saving an Image

The image saved does not need to be in the same format as the image read. For instance, **JPEG** doesn't have transparent colors, but I can write the image back out as a **PNG**, which does. The library has different functions for each image type.

Here's a simple example for the three formats which we're using:

```
stbi_write_png("pic.png", w, h, 4, data, w * 4);
stbi_write_jpg("pic.jpg", w, h, 4, data, 100);
stbi_write_bmp("pic.bmp", w, h, 4, data);
```

The first five arguments are the same for each type of output (although you call a different function name for each). Instead of **w** and **h**, you'll use the members **width()** and **height()**. The literal **4** is the channels (or components per color). We'll always use **4**. The number **100** is the quality of the JPG file.

Freeing the Memory

The ***stbi_load()*** function returns a pointer, but inside that function it asks the operating system to **allocate enough memory** to hold the image that it loads from disk. This memory is **on the heap**, which met in this chapter. In the C programming language, you have to remember to **free** that memory before your program ends. We do that by using the function ***stbi_image_free(paris)***.