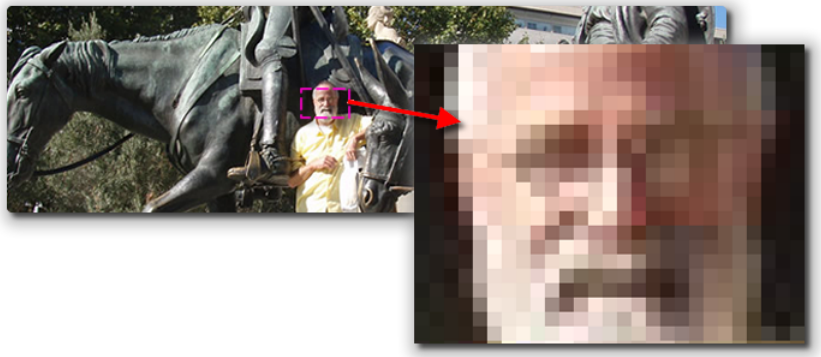
Digital Images

In this lesson we'll look at processing digital images. Here, for instance, is a picture that my wife took on vacation in Madrid, of me standing in front of the statue of Don Quixote and Sancho Panza.



Zooming in on my face, we can see that the image is actually made up of many square "**pixels**", each showing one color.

- Each pixel is a small square that shows a single color
- An 800 x 600 image is 800 pixels wide by 600 pixels high, which is 480,000 pixels in all (0.5 megapixels).
- The digital camera in your phone produces images with several megapixels per image. For instance, a 8000 x 6000 pixel image would be about 5 megapixels. The iPhone 13 camera is 12 megapixels

Every pixel has an `x,y` coordinate that identifies its location within the image grid.

RGB Colors

If every pixel is a single color, then how do we represent that color in memory? We use a scheme called **RGB**, or red, green blue. By combining different quantities of pure red, green and blue light, we can create any color at all.

Each of the red, green and blue light levels is encoded as a number in the range `0..255`, with `0` meaning zero light and `255` meaning maximum light.

So for example (`red=255`, `green=100`, `blue=0`) is a color where red is maximum, green is medium, and blue is not present at all, resulting in a shade of orange. In this way, specifying the brightness `0..255` for the red, blue, and green color components of the pixel, any color can be formed.

> *Pigment Note -- you may have mixed color paints, such as adding red and green paint together. That sort of "pigment" color mixing works totally differently from the "light" mixing we have here. Light mixing is easier to follow, and in any case, is the most common way that computers store and manipulate images.*

It's not required that you have an intuition about, say, what `blue=137` looks like. You just need to understand that any color can be made by combining the three color values.

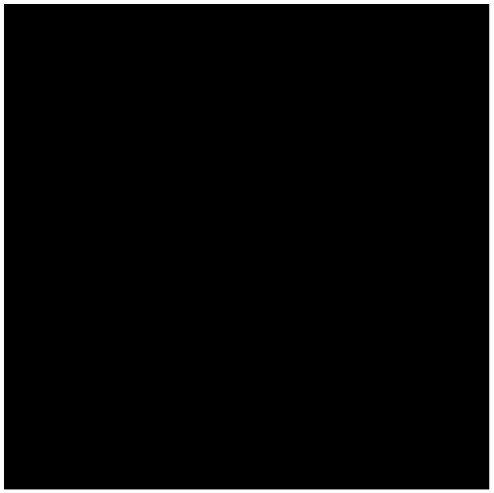
The RGB Explorer

red:

green:

blue:

`red:0 green:0 blue:0`



This pages lets you play with the RGB scheme, combining red, green, and blue light to make any color. The sliders control the red green and blue lights, each ranging from `0` (off) to `255` (maximum). The intersecting rectangles show the result of adding the red, green, and blue light together—any color can be created in this way.

To make pure red, green, or blue light, just turn up that color, leaving the other two at `0`. A few other common combinations:

- All at max (`255`) → white
- All at min (`0`) → black
- red + green → yellow
- red + blue → purple
- green + blue → turquoise
- Dark yellow—make yellow, then reduce both red and green
- Orange—make yellow, but more red, less green
- Light, pastel green—make pure green, then turn up both red and blue some equally (going towards white)
- Light gray—make white, then turn all three down a bit equally

Digital Images in C++

C++ itself doesn't have any built-in support for images, graphics or user-interfaces. All of those capabilities are added using libraries. Click on the little Running-Man in the right to open the **Replit** project for this lesson. Make sure you **Fork the Repl** to get your own copy to work on.

We're going to use the ***stb_image*** and ***stb_image_write*** libraries, written by Sean T. Barret (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.

The STB libraries are **header only** libraries. That means you only need to include the header file; there is no separate library to compile and link to. **I've already added the header files to the Repl you are working with in this lesson.**

In your own programs, https://github.com/nothings/stb download the latest version of the **`stb_image.h`** and **`stb_image_write.h`** from the GitHub site and place both files in your project folder. To make sure that the implementation is included, **in one file**, you need the **following preprocessor directives** before you include them:

```
#define STB_IMAGE_IMPLEMENTATION        // REQUIRED (loading)
#define STB_IMAGE_WRITE_IMPLEMENTATION  // writing
#include "stb_image.h"                  // "header-only" C libraries
#include "stb_image_write.h"
```

I've already added these to **`main.cpp`** in the Repl you are working with.

Loading an Image

When a digital image is stored on disk, it is compressed and encoded using a specific encoding scheme, such as **JPEG**, **PNG**, or **GIF**. The **stb** library function, `stbi_load()` will:

- Open the file and read and decode the image data
- Allocate enough memory to store the pixels **on the heap**
- Return a pointer to the **image data** as well as the **width**, **height** and **bytes-per-pixel** used to encode the original image.

More Loading an Image

The first part of the sample program (in `main.cpp`), loads a JPEG version of OCC's mascot, Pete the Pirate, into memory.

```
1  int width, height, bpp, channels = 4;
2  unsigned char * const pete =
3      stbi_load("pete.jpg",             // input file
4                &width, &height, &bpp,  // pointers (out)
5                channels);              // channels (in)
```

- The ***`stbi_load()`*** function returns a **pointer** to the first byte of the image data in memory. The type of the pointer is an **`unsigned char`**, which, in C++ speak means a "raw" byte. If loading fails, then the function returns the **`nullptr`**.

  Note that the pointer **`pete`** is a **`const`** pointer. This is necessary because you will later need to "free" the memory that the function has placed on the heap. If you move the pointer, then your program may crash.

- The first argument to the function is the **path to the file.** This can be absolute or relative (as used here), but it must be a **C-style string**. We'll look more at C-style strings in a future lesson. For right now, **use a literal** or use the **`c_str()`** member function on a regular C++ string.
- The next three arguments are the **address of the** **width**, **height**, and **bytes-per-pixel** used in the original image. These are **output parameters**; that means that you first create the variables (on line 1), and then pass **their addresses** as arguments. The function will **fill them in**. The information flows **out of the function**, not into it.
- The last argument is an **input** parameter telling the ***`stbi_load()`*** function how to store the image. Here we're telling it to store **4** bytes for each pixel (**RGBA**), even though the original image only has **3** (**RBG**).

Saving an Image

When we save the image, we can use a different format from the original. For instance, **JPEG** doesn't have transparent colors, but we can write the image back out as a **PNG**, which does. The ***`stb_image_write`*** library has different functions for each image type. Here's the code to save our image as **PNG**.

```
stbi_write_png("pete.png", width, height, channels, pete,
        width * channels);
```

Each file type you want to use has its own function, but the first five arguments are the same for each file type:

1. The file name as a C-style string. Here we've hard-coded **`pete.png`**
2. The **`width`** and **`height`** returned from ***`stbi_load()`***.
3. The number of channels (or bytes-per-pixel) used in memory to represent the image.
4. The pointer to the first byte of the image data in memory.

The last argument, **`width * channels`**, is unique to **PNG** files. It tells the function at what byte the next row begins.

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 you met in an earlier lesson. 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(pete)`***.

Changing the Format

Although the previous example added an alpha (transparency) channel to the Pete the Pirate picture, it didn't really change how it looked. Our second example loads this 4-channel **PNG** image as a 1-channel (that is, grayscale) image. We'll then save it as a **BMP** which is the native format for Windows applications.

Here's the code that that does this.

```
//Load a png file using 1 byte per pixel (Gray scale)
channels = 1;
unsigned char * const stego =
    stbi_load("stegosaurus.png", &width, &height, &bpp, channels);
stbi_write_bmp("stego-bw.bmp", width, height, channels, stego);
stbi_image_free(stego);
```

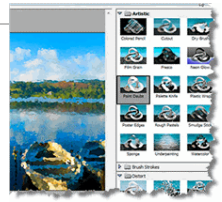
Notice that the conversion from 4-channel image on disk to 1-channel image in memory, happened when we **loaded** the image, not when we **wrote** the image.

PNG to JPEG

The first example changed a **JPEG** into a **PNG**, but the final example goes the other way, removing the alpha (transparency) channel and saving it as a **JPEG**. The ***stb_write_jpg()*** function also takes one extra argument, which is the quality of the resulting image. This can go from **0-100**, where **100** has the highest quality.

Go ahead now and type **make run** in the **Repl Shell**, and you'll be able to examine the modified images. In your homework, we'll use this newfound ability to read and write images to write **image filters**, like those found in PhotoShop.

Digital Filters



Last week you learned how to load and save digital images, using the functions in the **stb image libraries**. Now, let's make some changes to those images before you save them. Programs that do this are called **image filters**. You may have seen them in programs like Photoshop, or in the camera app on your phone.

Although the programs we'll write won't be as sophisticated or as fast as the commercial filters you can purchase, it will give you an idea about how such programs are written. (Plus, you'll get some exercise using pointers!)

Click the "running-man" to open the starter files in **ReplIt**. You'll have to Fork the **Repl** to get your own copy. When you open `main.cpp` you'll see some code similar to that you used last week. This time, we're going to load our picture of "Pete the Pirate" and modify it.

Address Arithmetic

When a pointer points to a contiguous list of data elements, such as the data stored on the heap by calling `stbi_load()`, we can apply the operators `+` and `-` to the pointer. This is called **pointer or address arithmetic**. Pointer arithmetic is similar to mixed type arithmetic with integers and floating-point numbers. If you add an integer and a floating-point number, the result is a floating-point number. Similarly:

- **Adding an integer** to a pointer gives us a **new address value.**
- **Subtracting one pointer from another** produces an integer.

Pointer addition considers the **size of the base type**; it doesn't just change the address by x number of bytes. Consider this code:

```
vector<int> v{1, 2, 3, 4, 5};
auto *p = &v[1];
cout << "p->" << p << ", " << *p << endl;
cout << "(p+1)->" << (p+1) << ", " << *(p+1) << endl;
```

When run, (click the previous link) you'll see something like this:

```
p->0x559a1c997eb4, 2
(p+1)->0x559a1c997eb8, 3
```

The pointer `p` contains the address `0x559a1c997eb4` (although it may be a different address when you run it), and it points to the second element in the `vector<int> v`. The address `(p + 1)` is `0x559a1c997eb8`. Note that for each unit that is added to a pointer value, the internal numeric value must be increased by the size of the **base type of the pointer**. In this case, that is `4` bytes, since the `sizeof(int)` is `4` on this platform.

Pointer Difference

Subtracting one pointer from another returns a **signed number** (of type `ptr_diff`) which represents the **number of elements** (**not** the bytes) between the two pointers. This is called **pointer difference**, and we'll use it more when we start looking at arrays.

Iterator Loops

Turning back to our image processing code, you can see that the pointer `pete` points to the first byte in our image once it is loaded into memory. Of course, **`pete`** is a **`const`** pointer, so it can't be changed. To process the image we need to create **a pair of pointers** like this:

- **`beg`** will be a non-const pointer which will move through all of the pixel data (using address arithmetic), so we can modify the image.
- **`end`** will be a **`const`** pointer that will contain the address just past the end of the data that **`stbi_load()`** has placed in memory. We can calculate this address also by using address arithmetic.

Here's the code you should add to **`main.cpp`** to create these two pointers.

```
unsigned char *beg = pete;    // beginning of the image
unsigned char * const end = pete + width * height * channels;
```

Notice that the expression **`width * height * channels`** is the total number of bytes in the image. By adding it to the pointer **`pete`**, we get a new address that is pointing at the first byte **following the image** in memory.

With these pointers, we can "visit" every byte in the image by using this **iterator loop**:

```
while (beg != end)
{
    // process the byte here
    beg++;    // move to the next byte
}
```

The Darken Filter

The first filter we'll write is "darken". Here's the algorithm:

Here's what the processing code looks like:

```
while (beg != end)
{
    if (*beg > 64) {
        *beg = *beg - 64;
    }
    beg++;
}
```

In the **Repl**, you can click on the two images to see the effect of applying the filter:

Dealing with Channels

Each individual pixel in our image consist of 3 bytes, each representing an individual red, green, or blue channel in that image. So, if you want to only modify one color, or two colors, you have to keep track of which byte you are working on, by processing **all three bytes every time through the loop**.

Here, for instance is a filter that only keeps the **blue** channel, and eliminates the **red** and **green** channels in the image:

```
while (beg != end)
{
   *beg = 0;   // turn off red
   beg++;      // go to next color
   *beg = 0;   // turn off green
   beg++;      // go to next color
   beg++;      // leave blue alone
}
```

Here's the result of running the blue filter:

State Filters

The darken and blue filters were both process filters: they applied the same rule to all of the pixels that they encountered. A **state** filter is one that looks for changes in the state of a pixel, such as it's location.

If we want to keep track of **where a pixel is located** in the image, you need to keep track of its position and perform an action when the state changes. Here's a filter that puts a white stripe on all of the even-numbered columns in a picture.

```
int x = 0;
while (beg != end)
{
   x++;               // go to the next column
   if (x % 2) {       // on odd columns
      *beg = 255;     // turn on red
      beg++;          // go to next color
      *beg = 255;     // turn on green
      beg++;          // go to next color
      *beg = 255;     // turn on blue
      beg++;
   }
   else  {  beg += 3; }  // don't do anything
}
```

Here's what the vertical-stripes filter looks like:

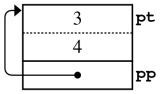
Pointers & Structures

We often use pointers in conjunction with structures or objects. Pointers are also used to work with the built-in C++ collection type, the **array**. We'll look at structures in the lesson, and arrays later.

Click the "running man" to visualize these statements, which create two variables.

```
Point pt{3, 4};
Point *pp = &pt;
```

The variable `pt` is a `Point` with the coordinates `3` and `4`. The variable `pp` is a pointer, pointing to `pt`. The memory diagram of these declarations looks like this. From `pp`, you move to the object by using dereferencing, so `*pp` and `pt` are synonyms.



If `pt` and `*pp` are effectively synonyms, you might expect to access `pt.x` by writing `*pp.x`. Surprisingly, **you cannot**. The expression `*pp.x` uses **two operators** so when you evaluate it, the **dot operator has higher precedence than the dereferencing operator**, so the compiler interprets the expression as `*(pp.x)`.

Of course, `pp` is a **pointer**, and that pointer **doesn't have** a member called `x`, so you get an error. Instead, you must write `(*pp).x` which is certainly awkward.

A (preferred) alternative, the operator `->` (usually read aloud as **arrow**), combines dereferencing and selection into a single operator. Knowing that, you can see there are three ways to print `x` and `y` in the variable `pt`:

```
1 | cout << "(" << pt.x << "," << pt.y << ")" << endl;
2 | cout << "(" << (*pp).x << "," << (*pp).y << ")" << endl;
3 | cout << "(" << pp->x << "," << pp->y << ")" << endl;
```

1. Line 1 uses a **structure variable** (an object) and the **member selection operation** (the "**dot**") operator, to select the members `x` and `y`.
2. Line 2 uses **the temporary structure object** obtained from **dereferencing** the pointer `pp`. That object is used with the member selection operator to select the same two variables, `x` and `y`.
3. Line 3 uses the pointer `pp` and the **arrow** operator to access the data members without first making a temporary copy.

Using the arrow operator is more efficient, and less typing, so you should use it when working with pointers to structures.

Using reinterpret_cast

The function `stbi_load()` always returns a pointer to the first byte of the digital image in memory, **not the first pixel**. That makes some code more complex than it needs to be. However, if you like, you can process your images **pixel-by-pixel** instead of **byte-by-byte** by following these instructions:

- Create a `Pixel` structure type with 3 `unsigned char` data members (since our image only uses 3 channels).
- When creating your `beg` pointer, change it to a `Pixel*`, not an `unsigned char *`, and then use `reinterpret_cast<Pixel*>` to cause `beg` to look at your image data pixel-by-pixel.
- When creating your `end` pointer, use `beg` as the base pointer. You no longer need to use `channels` as part of the calculation.

Here's a horizontal-stripes filter that does this, using **nested for loops** instead of an iterator loop, to change all the pixels in each odd numbered row to white stripe.

```
struct Pixel { unsigned char red, green, blue; };
Pixel * beg = reinterpret_cast<Pixel*>(pete);
Pixel * end = beg + width * height;

for (int y = 0;  y < height; ++y)
{
   for (int x = 0; x < width; ++x, ++beg)
   {
      if (y % 2 == 1) { // odd rows
          *beg = Pixel{255, 255, 255};
      }
   }
}
```

Here's what the horizontal-stripes filter looks like when it runs: