Chapter

18

Pointers & Graphics

You probably have used some sort of image-editing software, whether it is Photoshop, Illustrator, the GIMP or an online image editing tool. In this section you learn how to load and write PNG and JPG graphics files.

In addition to getting more practice working with pointers, you'll also get exposure to using external C open-source libraries. C++ itself doesn't have any built-in support for images, graphics or user-interfaces. All of those capabilities are added using libraries.

We're going to use 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.

Right-click the "running man" and open in a new tab to get started.

Using the STB 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 Sandbox you are working with in this lesson. In your own programs, download the latest version of the headers from the GitHub site:

> https://github.com/nothings/stb

Download these two files and place them in your project folder:

- **stb_image.h**
- **stb_image_write.h**

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"
```

Loading a JPEG Image

The first part of the sample program **loads** a JPEG version of OCC's mascot, Pete the Pirate, into memory.

```
// 1. Load a jpg file using 4 bytes per pixel (bpp RGBA)
int width, height, bpp, channels = 4;
unsigned char * const pete =
    stbi_load("pete.jpg",               // input file
              &width, &height, &bpp,    // pointers (out)
              channels);                // channels (in)
```

- The ***`stbi_load()`*** function returns a **pointer** to the first byte of the image in memory. The type of the pointer is an **`unsigned char`**, which, in C++ speak means "raw" byte. If loading fails, then the function returns **`0`**.
- Note that the pointer is a **`const`** pointer. This is necessary because you will later need to "free" the memory that the function returns. 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** of the image. These are **output parameters**; that means that we 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 the ***`Load()`*** function how to interpret the image. Here we're telling it to provide us with ***`4`*** channels of input (even though the original image only has 3-RBG).

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.

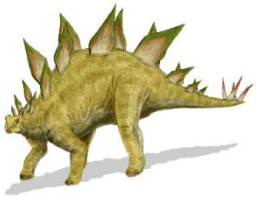
```
// Now write it out in current folder as a 4-bytes-per-pixel PNG
stbi_write_png("pete.png", width, height, channels, pete,
               width * channels);
```

The first five arguments are the same for each type of output (although you call a **different function name** for each). Notice how we're using the variables that were initialized by the previous function call. The last argument, **`width`** * **`channels`**, is unique to **PNG** files. It tells the function at what byte the next row begins.

Freeing the Memory

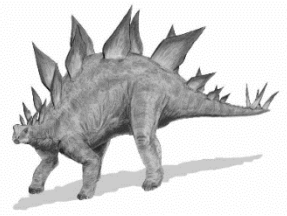
The ***`stbi_load()`*** function returns a pointer, but inside that function it asks the operating system to **allocate enough memory** to hold the image that it loads from disk. This memory is **on the heap**, which you met in an earlier lesson. In the C programming language, you have to remember to **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-color **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.



```
// 2. 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);
```

PNG to JPEG

The first example changed a **JPEG** into a **PNG**, but the final example goes the other way, removing the alpha 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.

You can run all three examples in the Sandbox shell by typing these two commands:

```
make main
./main
```

You can see the new files (and compare them with the originals) just by double-clicking them in the file explorer.

Pointers, Functions & Structs

When working with functions, you can simulate call by reference by using explicit pointer parameters. Many programmers prefer to do this because it is obvious that you are passing by reference instead of by value at the function call.

Let's look at a `swap()` function that exchanges the values contained in two integers. The algorithm is simple, assuming the two parameters are **a** and **b**.

```
Store the value of a in temp
Copy b into a
Copy temp into b
```

Using reference parameters, we write it like this:

```
void swap(int& a, int& b)
{
    int temp{a};
    a = b;
    b = temp;
}
```

When you call the function, there is no indication that **x** and **y** will be changed.

```
int x{3}, y{4};
swap(x, y);     // does this change x and y?
cout << "x->" << x << ", y->" << y << endl;
// x->4, y->3
```

To use explicit pointer parameters, change the implementation like this:

```
void swap(int *a, int *b)
{
    int temp{*a};
    *a = *b;
    *b = temp;
}
```

Then call it like this. The effect is the same, but you can tell at the call-site that this is a version of "pass-by-reference".

```
int x{3}, y{4};
swap(&x, &y);     // explicit address passing
cout << "x->" << x << ", y->" << y << endl;
// x->4, y->3
```

Pointers and const

Pointers have two values: an **indirect** and an **explicit** (or direct) value. Either or both may be `const`. Consider this code.

```
string a{"A"}, b{"B"}, c{"C"};
const string *ps1 = &a;
string * const ps2 = &b;
const string * const ps3 = &c;
```

Note the word `const` in the declaration of `p1`, `p2` and `p3`.

- Prevent **writing to the pointer's indirect value**, by putting the `const` **before the type** (`ps1`). Thus `*ps1 = string("x")` is illegal.
- When `const` comes **after the star**, (`ps2`), it means that the pointer itself cannot be changed; it **cannot make it point to a different location**. It would be illegal to write `ps2 = &a;`
- Prevent changing either the pointer, **or** what it points to, by using **both** versions of `const` (`ps3`).

When you write a function which **should not** change the item that it points to, make sure you define it with a `const` pointer. For instance, consider this template function which prints both the address and value of any variable:

```
template <typename T>
void printData(const T* p)
{
    cout << "Pointer: " << p << "->";
    if (p) cout << *p << endl;
    else cout << "nullptr" << endl;
}
```

Because the parameter is const, you can pass the function **both** `const` and non-`const` variables. It works with everything because it **guarantees** that it won't inadvertently change the object that `p` points to.

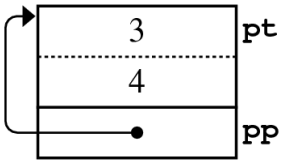
Pointers and Structures

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 in the next.

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 using dereferencing, so ***pp** and **pt** are synonyms.

If **pt** and ***pp** are effectively synonyms, you expect to access **pt.x** by writing ***pp.x**, since you can certainly access it by writing **pt.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 it 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**.

Pointer-To-Member Operator

Don't you find this 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**:

```
cout << "(" << pt.x << "," << pt.y << ")" << endl;
cout << "(" << (*pp).x << "," << (*pp).y << ")" << endl;
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 "**structure pointer dereferencing**" or **arrow** operator to access the data members.

You can run this yourself by clicking the link in this sentence.

Address Arithmetic

If a pointer points to **a contiguous list of data** elements, such as an element inside a `vector`, we can apply the operators + and – to pointers. This is called **pointer arithmetic**. Pointer arithmetic is similar to mixed type arithmetic with integers and doubles.

- **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**. 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->0x2cf2604, 2
(p+1)->0x2cf2608, 3
```

The pointer `p` contains the address `0x2cf2604` and it points to the second element in the `vector<int> v`. The address (`p + 1`) is `0x2cf2608`. 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`) that 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.

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using `make submit`.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.