# PART 8: "Addresses"

## Introduction

One of the harder concepts in C++ is the concept of *pointers*. Pointers have everything to do with the very nature of the language: C++ gets you pretty close to the hardware you are working with, and one of the most important parts of that hardware is memory. Your system probably has a few 'gigabytes' of it, which roughly corresponds to a couple of billion bytes:

1Kb = 1024 bytes,
1Mb = 1024 Kilobytes,
1Gb = 1024 Megabytes,
1Tb = 1024 Gigabytes,

…and so on. Note that 1024 is used rather than 1000, because 1024 is a power of 2 ($2^{10}$ in fact), and 1000 isn't. So, 1Gb of memory means that you can store about 1 billion unique values, or more precisely: $2^{30}$ unique values. Those values are numbered, and their location in the endless row of values is called an *address.* And, if you know at which address something is stored in memory, you can point to it.

## Back to the screen

Load up a fresh template project. Have a look at the function void Surface::Bar( int x1, int y1, int x2, int y2, Pixel c ) in surface.cpp:

```
void Surface::Bar( int x1, int y1, int x2, int y2, Pixel c )
{
	Pixel* a = x1 + y1 * m_Pitch + m_Buffer;
	for ( int y = y1; y <= y2; y++ )
	{
		for ( int x = 0; x <= (x2 - x1); x++ ) a[x] = c;
		a += m_Pitch;
	}
}
```

This function draws a bar. At the first line of this function, a *pointer* is created (denoted by the '`*`'). Once that line is executed, pointer `a` will contain the address of the first pixel of the bar. Then the function proceeds with drawing the actual pixels: it loops over the lines that contain the bar (`y1` to `y2`), and then over the rows that contain the bar (`x1` to `x2`). However, it does not use the Plot function: instead, it writes directly to the memory that contains the screen pixels.

So, let's do some experiments. Copy the following Tick function:

```
void Game::Tick( float deltaTime )
{
	Pixel* address = screen->GetBuffer();
	for ( int i = 0; i < 307200; i++ ) address[i] = i;
}
```

This code gets the address of the screen surface in memory, and stores it in a pointer variable. Then, it uses a single for-loop to fill (most of) the screen. Now try this code:

```
void Game::Tick( float deltaTime )
{
	Pixel* address = screen->GetBuffer() + 100;
	for ( int i = 0; i < 255; i++ ) address[i * 800] = i;
}
```

This gets us a vertical line. This shows something important:

*The lines of your screen may appear separate, but they are not: in memory, your screen is one continuous line, consisting of 800 * 512 pixels.* That means that you should look 800 pixels further to get to the next line. Which is exactly what above code does. This also explains why printing text too close to the right edge of your window makes it wrap to the left side: *there is no right edge.*

### Variables in memory

Let's leave the screen pixels for a moment. Copy the following Tick function:

```
void Game::Tick( float deltaTime )
{
	int a = 100;
	int b = 200;
	int* c = &a;
	*c = 300;
	int w = 0;
}
```

Put a breakpoint on the third line (`int* c = &a;`), and run the code. Your program will stop right before it executes that line. Now, open a 'watch' window, and inspect the following values:

`a` (should be `100` of course);
`&a` (will be some `0x00abcd01` number).

The '`&`' sign, when attached to a variable name, gives you the *address* of a variable. So in this case, `a` is stored somewhere in memory, and now you know exactly where.

The same symbol is used in the third line of the code. Here we create a pointer (notice the '`*`' symbol), which points to `a`. So there is a fundamental difference between variable `a` and `c`: `a` is an integer, but `c` is a *pointer to an integer*, i.e., it contains the address of something that is supposed to be an integer.

### But… Why!?

So why are pointers so important in C++? The reason is: performance. When you plot a pixel at coordinate (`x`, `y`), in the end all that matters is that some value at some memory location changes. That's a simple operation, in principle. But when you only have `x` and

`y`, you need to calculate that address: it's on line `y`, so at least `800 * y` pixels past the start of the screen start in memory. Add `x` to that to get the correct address. O, and don't forget the address of the screen itself. So the formula is: `screen->GetBuffer() + x + y * 800`. At that point you just have the address, so you still need to plot. That's an awful lot of work for a single pixel. If you *know* the address, all you need to do is put a value right there. Pointers give you low-level control over memory. This gives you fine control over the work that is carried out, and this in turn gives you raw performance (and thus, speedy games).

**Assignment**

Here's your task for today:

1. Draw a dotted line (skip every other dot) from (0, 0) to (400, 400). Do not use the Plot function, use a pointer variable instead. Figure out what the distance between dots *in memory* is to make this code super simple and fast.
2. Create an application that does the following:
    a. Load an 800x512 image
    b. Moves a white dot from the top-center (say, x = 400, y = 0) to the bottom. While doing so, the dot evades obstacles.
    c. When the pixel below the dot is black, it goes down. Otherwise, it goes one pixel to the left if its y is an odd number, or the right if its y is an even number.
    d. When the pixel reaches the bottom of the screen, it returns to the top.

   Make sure that the image that you use for step *a* is suitable for the other steps. Also note that even though we're moving a single pixel now, this will be an avalanche soon. ☺ For this assignment, you need to use pointers: it's also a great way to *read* screen pixels rather than writing to them.

Once you have completed this assignment, you may continue with the next part.