On Degeneration of Array Types in C

Paul R. Potts

March 2015 and January 2025

This content started out as the last draft of a post from 2015, which I apparently never published, in my now-retired Blogger blog, "Praise, Curse, and Recurse." In January 2025, I expanded and improved it and added illustrations.

Note that if you want to play with the source code I used to write this article, you can find the GitHub repository here. I have also included links below at the end of each section, referring to the specific C file containing code from that section.

Years ago, I came to think of C as the language that takes everything you've carefully told it about your variables, and then *throws most of it away.* Working recently in Haskell, it's been great to see a language that *doesn't* do that. This has inspired me to I want to think a little on the subject of typing as it applies to array types. Many programmers, even many C programmers, don't seem to fully grasp what C is doing (and not doing) for them when they use array types. Some people even go so far as to say that C is a very simple language. While it has a small set of keywords and standard types, it isn't actually very simple in practice. So let's talk about that.

It's a commonplace that arrays and pointers in C are somehow equivalent. But that statement, in and of itself, is wrong in several important ways. Let's say you have a one-dimensional array of enumerations:

```
typedef enum {
    Red, Orange, Yellow, Green, Blue, Indigo, Violet
} color_t;

#define DIM_X 5

typedef color_t colors_one_d_a_t[DIM_X];

colors_one_d_a_t colors_one_d_a = {
    Green, Indigo, Red, Orange, Yellow
}
```

Here's a representation of the array showing its indices:

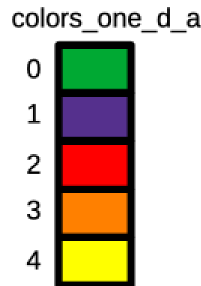

Figure 1: “A One-dimensional Array of Colors”

Note that I'm using my own scheme for encoding types in typedef'ed names, a bit like Hungarian Notation. In this case, I'm defining **colors_one_d_a_t** as the type of a one-dimensional array, then making a variable of that type, colors_one_d_a, and initializing it. This is overkill for our simple example, but I developed this style when when writing code containing structures, arrays, structures as array elements, arrays as structure elements, etc., and it can be helpful for keeping track of what I can do with an object.

We can then access members of the array in the usual way:

```
printf( "first 1-D array element: %d\n", colors_one_d_a[0] );
```

I get:

```
first 1-D array element: 3
```

and that's because enums start at zero and, unless you specify otherwise, are monotonically increasing integral types, thus, the **Green** enumeration has the value 3. Let's look at the addresses now, of the array and of the first element:

```
printf( "address of 1st element: %p\n", &colors_one_d_a[0] );
printf( "address of whole array: %p\n", colors_one_d_a );
```

I get:

```
address of first element: 0x100001020
address of whole array: 0x100001020
```

Of course, your addresses will probably be different.

That address of the first element should be uncontroversial; there's no prefix or suffix or other overhead in a C array, which consists only of data elements laid out in consecutive memory locations, with no gaps, so the address of the first element is the same as the address of the whole array. But — and this is more important — just why we can pass *the array variable itself* to **printf()** and print

it with the **%p** (pointer) format specifier? The compiler happily accepts it and returns the address of the array. Why isn't that a type error?

A Bunch of Degenerates

It's not a type error because in C, array types are not really full, first-class types that can be used and passed around like any other type. In most circumstances, when we want to *use* an array, whether by accessing accessing elements, or passing it to a function — the array type "degenerates" or "decays" into the type of pointer to the array's elements, with the value of a pointer to the first element. So, in this case, **printf** sees a parameter of type **color_t ***, *not* a parameter of type **colors_one_d_a_t**.

Why? Well, for that, you can blame Dennis Ritchie's goal of maintaining as much compatibility as possible with B and BCPL, precursors to C; see his paper The Development of the C Language:

> The rule, which survives in today's C, is that values of array type are converted, when they appear in expressions, into pointers to the first of the objects making up the array.

It also makes it so that passing an array to a function was never done by making a copy of the array on the stack, which is one of the reasons that C code runs so efficiently.

This might seem like I'm arguing some kind of trivial point or useless technicality, but I assure you, it isn't, as we'll see — it has important implications.

Now, let's try printing out all the elments:

```
printf( "elements: %d, %d, %d, %d, %d\n",
    colors_one_d_a[0], colors_one_d_a[1],
    colors_one_d_a[2], colors_one_d_a[3],
    colors_one_d_a[4] );
```

This gives me: elements: 3, 6, 0, 1, 2

Let's verify how this is implemented. Here's the disassembly (on an Intel Mac):

```
movl _colors_one_d_a(%rip), %esi
movl _colors_one_d_a+4(%rip), %edx
movl _colors_one_d_a+8(%rip), %ecx
movl _colors_one_d_a+12(%rip), %r8d
movl _colors_one_d_a+16(%rip), %r9d
movl %eax, -24(%rbp) ## 4-byte Spill
movb $0, %al
callq _printf
```

Even if you didn't quite understand that disassembly, the basic idea should be clear: **colors_one_d_a** is a *memory address*. The code retrieves the values at **colors_one_d_a**, the address of the first element in the array, and then at

colors_one_d_a + 4, the address of the second element in the array, etc., continuing through **colors_one_d_a + 16** until it has stored the values of all five elements of the array. Where we're storing them isn't really important for purposes of this discussion; the important thing is that **printf** knows where to retrieve them, because this code follows the C calling convention for this platform, which means that it stores the parameters in places where **printf** knows to look for them. But the other important thing to note is that the addresses are calculated *at compile time*. They are baked into the code.

So, now let's declare a explicit pointer variable, and access the same array elements with a pointer, using array notation with the pointer:

```
color_t * color_p = colors_one_d_a;
printf( "elements: %d, %d, %d, %d, %d\n",
    color_p[0], color_p[1], color_p[2],
    color_p[3], color_p[4] );
```

That prints out exactly the same values. The fact that it does should be non-controversial — using an explicit pointer to access elements in an array is a very common C programming idiom. In fact, as we'll see later, the square brackets are really just "syntactic sugar" for pointer math! But it's *not* working because arrays and pointers are exactly the same thing and work exactly the same way. By introducing a variable in the form of a pointer, we've introduced explicit indirection, and the compiler must consider that the value of color_p could *change*, while the value of *colors_one_a_d*, which decays into a pointer to the arrays first element when used in an expression, is *immutable*.

Let's take a look at the compiled code:

```
movq    -24(%rbp), %r10
movl (%r10), %esi
movq -24(%rbp), %r10
movl 4(%r10), %edx
movq -24(%rbp), %r10
movl 8(%r10), %ecx
movq -24(%rbp), %r10
movl 12(%r10), %r8d
movq -24(%rbp), %r10
movl 16(%r10), %r9d
movl %eax, -36(%rbp) ## 4-byte Spill
movb $0, %al
callq _printf
```

That looks — different! The pattern for the array element access is to look up *the address stored in our pointer* and put it in a register, then read the array values from memory locations, by using offsets from that base address (again, +0, +4, +8, +12, and +16).

In other words, every array element is now accessed *indirectly* by adding an

offset to a pointer variable, and those memory addresses where the first five elements of our array are stored are now calculated *at runtime*, not baked into the code (the offsets are baked in, but adding the offsets to the base address in the pointer, stored in register **r10**, is done at runtime, then the looked-up array values are stored in various registers prior to calling **printf**, the same way as before). Here's a drawing of the situation:

Figure 2: "A Pointer to a One-dimensional Array of Colors"

Read the previous paragraphs again if you don't understand this. Make this into a post-it note and stick it on your forehead. It's important! Using the array subscript operator (the square brackets) on a pointer type generates *very different* code than using the same array subscript operator on an array type!

Hit the Books

You might have seen the degeneration of array types discussed in books like *C Programming FAQs* by Steve Summit, or *C Traps and Pitfalls* by Walter Koenig, or *Expert C Programming: Deep C Secrets* by Peter van der Linden. I especially recommend that last one! But I've met experienced C programmers who don't fully understand it. Failure to understand this distinction can cause you no end of pain, especially when passing arrays between functions, or referencing arrays that are defined in one file and referenced from another file using an **extern** declaration, or trying to work with dynamically allocated memory the same way you would work with statically allocated arrays — but I'll come back to that.

Here's that "degeneration" in action. In the example of direct element access above, I was using a **printf** statement in **main()** and referring to an array declared at file scope. Let's assume that array is still there, but pass it to a function to do the exact same thing:

```
void print_elements_a( color_t colors_a[] )
{
    printf( "elements: %d, %d, %d, %d, %d\n",
        colors_a[0], colors_a[1],
```

```
        colors_a[2], colors_a[3],
        colors_a[4] );
}

int main(int argc, const char * argv[])
{
    printf( "elements: %d, %d, %d, %d, %d\n",
        colors_one_d_a[0], colors_one_d_a[1],
        colors_one_d_a[2], colors_one_d_a[3],
        colors_one_d_a[4] );

    print_elements_a( colors_one_d_a );

    return 0;
}
```

Note that the parameter for the **print_elements_a** function, **color_t colors_a**[] is really just a pointer in disguise. The empty square bracket syntax lets us indicate to the reader that the function is designed to be called with an array, not a single pointer. As the compiler has thrown away the array length, it is not required to generate an error or warning if you access elements that are outside the bounds of the array. Note that you *can* provide a length, like so:

```
void print_elements_a_with_length( color_t colors_a[DIM_X] )
{
    printf( "print_elements_a: elements: %d, %d, %d, %d, %d\n",
        colors_a[0], colors_a[1],
        colors_a[2], colors_a[3],
        colors_a[4] );
}
```

But it makes no difference; if I add another **%s** to the format string and add **colors_a[5]** to the list of parameters, which accesses an element outside the bounds of the array, my compiler doesn't generate a warning or error. We've just entered the territory of Undefined Behavior, which is bad. This is one of the most common examples of why C is considered an unsafe language.

Anyway, the code works exactly the same way if we write it like this, explicitly specifying a pointer, rather than using that array notation which degenerates into a pointer:

```
void print_elements_a(color_t * colors_a_p)
{
    printf( "elements: %d, %d, %d, %d, %d\n",
        colors_a_p[0], colors_a_p[1],
        colors_a_p[2], colors_a_p[3],
        colors_a_p[4] );
}
```

In **main**, the generated code uses the direct offset form, while in **print_elements_a**, the generated code uses the *indirect* offset form — on every element lookup — because the array type degenerates to a pointer-to-element type.

Note that if you want to play with the source code I used to write this part of the article, you can find it on GitHub here.

Moving On to Arrays of Arrays

OK, so this may seem relatively trivial for a 1-dimensional array, but what happens with a 2-dimensional array? In C, we don't *really* have multi-dimensional array types the way other languages do, but we can implement *arrays of arrays*, and if we use multiple sets of square brackets on an array type, the compiler will do the math necessary to access the array elements *as if* it was a real two-dimensional array type that carried inside it information about its number of dimensions, bounds, etc. But again, when you use the name of the array in an expression, it degenerates to a pointer type, and when that happens, it becomes much less usable. I'll go into detail below.

I've shown how this degeneration has important implications for the generated code when working with one-dimensional array types. The difference is even more stark when working with two-dimensional (or higher-dimensional) array types. Let's look at a simple two-dimensional array:

```
#define DIM_X 5
#define DIM_Y 7

typedef color_t (colors_two_d_a_t)[DIM_Y][DIM_X];

colors_two_d_a_t colors_two_d_a = {
    { Violet, Violet, Violet, Violet, Violet },
    { Indigo, Indigo, Indigo, Indigo, Indigo },
    { Blue, Blue, Blue, Blue, Blue },
    { Green, Green, Green, Green, Green },
    { Yellow, Yellow, Yellow, Yellow, Yellow },
    { Orange, Orange, Orange, Orange, Orange },
    { Red, Red, Red, Red, Red }
};
```

The compiler lays these elements out in successive memory locations without holes or padding, as if we were really allocating a 2-dimensional array of size **DIM_Y * DIM_X**. This is *row-major order*:

Here's a statement that does 2-D lookup of array elements:

```
printf( "elements: %d, %d, %d, %d, %d\n",
    colors_two_d_a[0], colors_two_d_a[1],
    colors_two_d_a[2], colors_two_d_a[3],
    colors_two_d_a[4] );
```

Figure 3: "A Two-dimensional Array of Colors"

Let's compile this, and then let's look at the assembly again.

Oh, wait, that's wrong. (Yep, I made the code wrong deliberately). We've tried to access elements as if this was a one-dimensional array, not a two-dimensional array. A set of square brackets is missing. But the error might not be the one you'd expect. My compiler says:

> Format specifies type 'int' but the argument has type 'color_t *'

It does *not* say anything about failing to use the right number of sets of square brackets to acccess an element. Why? Because our pseudo-2-dimensional array really *is* treated like an array-of-array. When we used **colors_two_d_a** in an expression, its type degenerates into the type of a pointer to its first element, which has the type of a one-dimensional array. Using the expression **colors_two_d_a[0]** yields an object whose type is a 1-dimensional array. And then when we pass *that* object as a parameter to **printf**, its type degenerates to the type of a pointer to the array's first element, with the type ___color_t *___. Got that?

Now let's fix our error and access the elements shown:

```
printf( "elements: %d, %d, %d, %d\n",
    colors_two_d_a[0][0], colors_two_d_a[1][1],
    colors_two_d_a[6][3], colors_two_d_a[6][5] );
```

Oh, wait, that's still wrong. I introduced an another error! When I tested this code originally using Clang and a demonstration project in Apple's XCode IDE, I got this warning:

> Array index 5 is past the end of the array (which contains 5 elements).

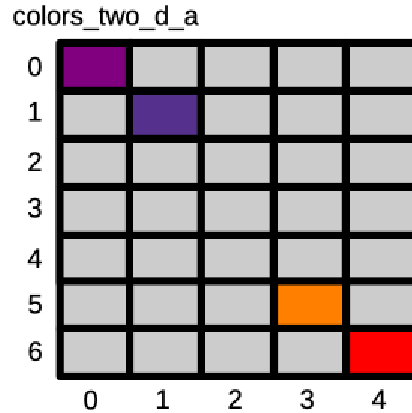

Figure 4: "Accessing Elements of a Two-dimensional Array"

But note that when I compiled it using **cc** on Ubuntu, I saw no errors or warnings. This situation can very likely be improved by enabling compiler warnings. But let's change that 5 to a 4:

```
printf( "elements: %d, %d, %d, %d\n",
    colors_two_d_a[0][0], colors_two_d_a[1][1],
    colors_two_d_a[6][3], colors_two_d_a[6][4] );
```

The assembly looks like this:

```
movl _colors_two_d_a(%rip), %esi
movl _colors_two_d_a+24(%rip), %edx
movl _colors_two_d_a+112(%rip), %ecx
movl _colors_two_d_a+136(%rip), %r8d
```

It's using direct offsets from the base address of the array, and it's calculated them out in row-major indexing — multiplying the higher-order row (y) index by **DIM_X** to figure out how far to skip ahead for each row, then adding the lower-order column (x) index times the size our elements (in this case, 4). The offsets are calculated like so:

```
( 5 * 0 + 0 ) * 4 = 0
( 5 * 1 + 1 ) * 4 = 24
( 5 * 5 + 3 ) * 4 = 112
( 5 * 6 + 4 ) * 4 = 136
```

Stride Lengths

Because we're using the array type *before* it has degenerated, the array indices are known at compile time, which means the *stride lengths* are known. The

stride lengths are the sizes of the rows and the sizes of the elements that the code has to "stride" (step) over, to get to the elements it wants. The stride length us used to calculate offsets at compile time, which means they don't need to be calculated at runtime. They are embedded right in the generated code.

Got that?

A Spoonful of Sugar

It's time to talk about how array access notation, those square brackets, are really "syntactic sugar" for pointer math. Per Wikipedia,

> The subscript notation **x[i]** (where *x* designates a pointer) is syntactic sugar for ___*(x + i)___. Taking advantage of the compiler's knowledge of the pointer type, the address that **x + i** points to is not the base address (pointed to by *x*) incremented by *i* bytes, but rather is defined to be the base address incremented by *i* multiplied by the size of an element that *x* points to. Thus, **x[i]** designates the _i+1_th element of the array.

Whew. I hope you're still following along! Note that this suggests that when we do explicit pointer math, we *don't need* to include the stride length, because C's pointer math knows about the pointer type and uses it to multiply *i* by the size of the element.

Explicit Pointers

Now let's look at pointer access to array elements. Let's make a pointer variable to access the elements of our array of arrays:

```
color_t * color_two_p = colors_two_d_a;
```

Oh, wait. I get a warning here!

> Incompatible pointer types initializing 'color_t *' with an expression of type 'colors_two_d_a_t' (aka 'color_t [7][5]')

(I'm attempting to teach this subject using the "make the compiler explain what is wrong" method, so you will learn to do it right! Never ignore your compiler errors or warnings, by the way!)

Again, this is a smart compiler! The type of this array, when it has "degenerated" to a pointer to its first element, is not actually **color_t *** or even **color_t ****. Remember, we have an *array of arrays.* The type of the first element is our one-dimensional array type, **colors_one_d_a_t**. So the array type in this context "degenerates" to a pointer to that type: **colors_one_d_a_t ***.

If we correct our variable definition, like so:

```
colors_one_d_a_t * color_two_p = colors_two_d_a;
```

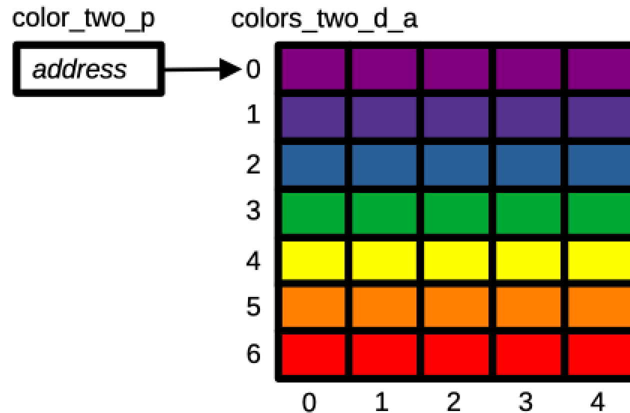

Figure 5: "Accessing Elements of a Two-dimensional Array Indirectly"

We now have this situation:

If we use the array subscript operator on that type to print out some elements, like we did above:

```
printf( "elements: %d, %d, %d, %d\n",
    color_two_p[0][0], color_two_p[1][1],
    color_two_p[6][3], color_two_p[6][4] );
```

we acccess the same elements, but we access them indirectly. Here's the generated code:

```
movq -32(%rbp), %r10
movl (%r10), %esi
movq -32(%rbp), %r10
movl 24(%r10), %edx
movq -32(%rbp), %r10
movl 112(%r10), %ecx
movq -32(%rbp), %r10
movl 136(%r10), %r8d
movl %eax, -52(%rbp)          ## 4-byte Spill
movb $0, %al
callq _printf
```

Note that the multiplied-out constants look the same. The compiler has combined the dimensions into a 1-D offset, but it is an *indirect* offset each time, via the pointer.

Let's look more closely at how the last parameter to **printf**, **color_two_p[6][4]**, works. The variable **color_two_p** has the type ___colors_one_d_a_t *___.

It’s just like our previous example where we made a pointer to a one-dimensional array:

Figure 6: “A Pointer to a One-dimensional Array of Colors”

We can make use of that “semantic sugar.” We’re telling the compiler that there are in fact at least seven rows. The expression **color_two_p[6]** will point to the element (which we are telling the compiler is there) at the start of the seventh row. The compiler at this point knows the size of the underlying element type, and the number of elements in the one-dimensional row array type. So it can do the pointer math to find the address of the start of the seventh row using the pointer value + (6 * 20) or 120. The row part is now taken care of. The next set of square brackets applies the pointer math to adjust this further, using the element size, not the row size, calculating 120 + 4 * 4, to stride past the first four elements of the seventh row. So we have our familiar 136, which is precalculated, but the math to apply that offset is done at runtime, not compile-time.

This *semantic* difference between array types and pointer types, combined with *syntactic* similarity, enables some useful coding idioms. For example, we can easily do 1-dimensional element lookup in a piece of memory that was allocated with **malloc**, or passed in to any old function, as if it was an array. We don’t have to use some alternate syntax like **->**[]. But note that the generated code is more complex — and might perform more poorly. And that “syntactic sugar” can *hide* the indirect access so that it might not be immediately obvious, when reading the code, that it is less efficient than code that used array types that hadn’t “degenerated.” (Another reason to maintain some information in the variable names!) And in addition, the array type degeneration that happens when we pass an array type as a parameter can make all but simple one-dimensional arrays much more difficult to use.

Note that if you want to play with the source code I used to write this part of the article, you can find it on GitHub here.

Using Pointers to Access Multi-dimensional Array Elements in Dynamically-allocated Memory

So what if you want to treat a **malloc**-allocated block of memory as a 2-dimensional array (really, an array of arrays) using the square bracket "syntactic sugar?"

Basically, you can't do it!

Well, you *can*, but the "degeneration" of array types sure does make things harder than you might initially expect.

When you're working with a pointer type, the compiler doesn't have the stride lengths of the *higher-rank types*, which would tell it how far to walk for each row (or higher dimension). And so you might find yourself needing to do your 2-D, or higher, access by doing your own for a 1-D access — using the row index * the x-dimension plus the column index. C's pointer math, which takes care of the element size automatically, helps here.

This obviously gets ugly in higher dimensions! But it also forces your code to repeat itself — you'll have to specify the dimensions in your explicit pointer math. This is why I use **#define** macros to specify my row lengths and number of columns; it allows me to change the values in only one place.

It seems like we ought to be able to allocate a block of memory and then somehow convince the compiler that it has our array type. Surely there's a way, right? So let's try it: we'll allocate some raw memory, and then create an array variable that references it by casting the pointer to the allocated memory:

```
void * raw_mem_p = malloc( DIM_Y * DIM_X * sizeof ( color_t ) );
colors_two_d_a_t colors_two_d_a = (colors_two_d_a_t)raw_mem_p;
```

Err, no, we can't use an array type like that at all — we can't have somehow turn a pointer into a *reference* to an array. The short answer is that when we have that pointer to raw memory, we have an *address*, not an "object" (in the C language sense, not the C++ language sense). My compiler says an "arithmetic or pointer type is required." I'm the kind of geek who likes to read language standards, but it would involve going down a deep rabbit hole to explain this further.

Anyway, what happens if we cast the allocated memory pointer to a pointer-to-array type, not an array type? That's just casting one pointer type to another, right?

```
colors_two_d_a_t *colors_two_d_p =
    (colors_two_d_a_t *)raw_mem_p;
```

It's legal to create that pointer, but that pointer type makes it *less useful than one might hope*. For example, we might imagine that if we have a pointer to memory containing an array, and an object of the same array type, that we could copy the array to the memory containing the array like this:

```
*colors_two_d_p = colors_two_d_a;
```

But no, that is not allowed. I get an error that says "array type... is not assignable." Why can't we do this? It's because when we mention the name of the array on the right-hand side of the expression, the type of that name degenerates to the type of the array's *elements*. This means that the compiler sees the code as if it were written like this:

```
*colors_two_d_copy_p = (colors_one_d_a_t *)colors_two_d_a;
```

So we'd be trying to write a pointer into our array of raw memory. We're treating it as containing color elements, not pointers to our one-dimensional array types. Even if it were legal to assign arrays like this, that would be a type error.

But, also, keep in mind — C simply does not allow array types be assigned to like this, while it does allow assignment can be done with structures (although this feature should be used with caution, especially if the structure contains pointers, as this results in simply copying pointers, which may not be what you want).

We also can't copy a row at a time like this:

```
for ( unsigned int y_idx = 0; y_idx < DIM_Y; y_idx++ )
{
    colors_two_d_p[y_idx] = colors_two_d_a[y_idx];
}
```

or copy our elements using nested loops:

```
for ( unsigned int y_idx = 0; y_idx < DIM_Y; y_idx++ )
{
    for ( unsigned int x_idx = 0; x_idx < DIM_X; x_idx++ )
    {
        colors_two_d_p[y_idx][x_idx] =
            colors_two_d_a[y_idx][x_idx];
    }
}
```

The problem here is that the pointer on the left-hand side, **colors_two_d_p**, with type **colors_two_d_a_t *** is a pointer to a *whole array*, not an element type — and so is pretty much *useless*.

Hold on, what? Can't we use the square brackets to access elements in it?

No, we can't!

Because the type of this pointer is that of a pointer to the *whole array of arrays*! The stride length that will be used, when we append square brackets to access elements, is the size of the whole array-of-arrays array type. If we evaluated **colors_two_d_p[1]**, the resulting pointer would point to the start of a *non-existent second array-of-arrays* of type **colors_two_d_a_t** (which, if it actually existed, would sit in memory just after to the real **colors_two_d_a**).

Wait, really?

Yes, and we can prove it. Let's write a bit of test code:

```
colors_two_d_a_t * colors_two_d_p = (colors_two_d_a_t *)raw_mem_p;
void * array_of_arrays_1_p = colors_two_d_p[0];
void * array_of_arrays_2_p = colors_two_d_p[1];
printf("diff: %td\n", array_of_arrays_2_p - array_of_arrays_1_p);
```

Note that since I'm using ___void *___ types here, the difference of pointers is not calculated in units of the size of the array elements. We get the difference in bytes.

I get 140, which is the size of our whole array-of-arrays, which contains 7 rows of 5 columns of 4 bytes each. Accessing anything past row zero would access memory that isn't part of the array-of-arrays. That's undefined behavior, and it's Very Bad — code like this might crash, which has the benefit of at least alerting us that something has gone wrong. But it might not crash. It might wind up corrupting memory, or creating an exploitable security hole.

Let's say we wanted to do it like this, instead, creating pointers in our loops:

```
for ( unsigned int y_idx = 0; y_idx < DIM_Y; y_idx++ )
{
    colors_one_d_a_t *colors_one_d_copy_p =
        &colors_two_d_copy_p[y_idx];

    for ( unsigned int x_idx = 0; x_idx < DIM_X; x_idx++ )
    {
        color_t *color_copy_p = colors_one_d_copy_p[x_idx];
        *color_copy_p = colors_two_d_a[y_idx][x_idx];
    }
}
```

Let's instrument that function and leave out the actual writes to memory, to see what stride lengths are being applied:

```
colors_two_d_a_t * colors_two_d_p = (colors_two_d_a_t *)raw_mem_p;
void * colors_one_d_prev_p = &colors_two_d_p[0];

for ( unsigned int y_idx = 0; y_idx < DIM_Y; y_idx++ )
{
    colors_one_d_a_t * colors_one_d_p = colors_two_d_p[y_idx];
    printf("diff of row pointer: %td\n",
        (void *)colors_one_d_p - colors_one_d_prev_p);
    colors_one_d_prev_p = colors_one_d_p;
    void * color_prev_p = &colors_one_d_p[0];
    for ( unsigned int x_idx = 0; x_idx < DIM_X; x_idx++ )
    {
        color_t *color_p = colors_one_d_p[x_idx];
```

```
            printf("diff of element pointer: %td\n",
                (void *)color_p - color_prev_p);
            color_prev_p = color_p;
        }
    }
```

When we do this, we find that, indeed, the code that is supposed to be striding by a *row* is striding by a whole array-of-arrays (140 bytes), and the code that is supposed to be striding by an *element* is striding by a whole row (20 bytes).

Using Explicit Pointer Math

So what's the workaround? Basically, we have to make our pointer types refer to the type of the elements, not the array-of-arrays or the row arrays, and do our own pointer math. But, we can still use the square bracket syntactic sugar as much as possible, to try to make it clear to the reader what we are doing:

```
void * raw_mem_p = malloc( DIM_Y * DIM_X * sizeof ( color_t ) );
color_t * colors_two_d_p = (color_t *)raw_mem_p;
printf( "dest base address: %p\n", colors_two_d_p);

for ( unsigned int y_idx = 0; y_idx < DIM_Y; y_idx++ )
{
    for ( unsigned int x_idx = 0; x_idx < DIM_X; x_idx++ )
    {
        // Create the element offset explicitly
        colors_two_d_p[y_idx * DIM_X + x_idx] =
            colors_two_d_a[y_idx][x_idx];
    }
}
```

For the assignment, note that this code, which avoids square brackets on the left side, is equivalent:

```
*( colors_two_d_p + y_idx * DIM_X + x_idx ) =
    colors_two_d_a[y_idx][x_idx];
```

This code is not equivalent, though: it has the same stride problem we discussed above, and will not work:

```
*( colors_two_d_p + y_idx * DIM_X + x_idx ) =
    *( colors_two_d_a + y_idx * DIM_X + x_idx );
```

although it would be nice if such a symmetrical approach would work. You can make it work like this:

```
*( colors_two_d_p + y_idx * DIM_X + x_idx ) =
    *( *colors_two_d_a + y_idx * DIM_X );
```

or by defining a pointer:

```
color_t * dest_color_a_p = *colors_two_d_a;
```

which can then be used in the symmetrical approach to hide the extra dereference needed:

```
*( colors_two_d_p + y_idx * DIM_X + x_idx ) =
    *( dest_color_a_p + y_idx * DIM_X );
```

However, I recommend just sticking with the first version, which uses square brackets as much as possible.

Note that if you want to play with the source code I used to write this part of the article, which includes more logging and testing code, you can find it on GitHub here.

Half-baked Array Types

And so, the half-baked array types in C sometimes *don't buy us much* and we can't really use them the way we'd like to when passing arrays around or casting allocated memory to an array type.

There's more to learn about array types — like the way that higher-dimensional arrays lose a dimension when they degenerate. I'm not going to go into all that now. But I do have a couple more things to show you.

Working with Dynamically-allocated Pointers

It might seem like it is pretty much impossible to try to use multi-dimensional array indexing operations with pointers rather than true arrays which were allocated as arrays. But that's not entirely true. There's a trick you can do. It is kind of horrific — but if you are really dedicated to using C's array syntax to hide pointer math, even though it obscures what is going on with the indirect offset indexing, the way that pointer types degenerate when you apply array indexing can actually help you. Check this out: this code will allocate an array of pointers, and then set each pointer in that array to a newly allocated array of elements.

```
color_t ** colors_two_d_pp =
    (color_t **) malloc( DIM_Y * sizeof( color_t * ) );
for ( unsigned int y_idx = 0; y_idx < DIM_Y; y_idx++ )
{
    colors_two_d_pp[y_idx] =
        (color_t *)malloc( sizeof( colors_one_d_a_t ) );
    for ( unsigned int x_idx = 0; x_idx < DIM_X; x_idx++ )
    {
        colors_two_d_pp[y_idx][x_idx] =
            colors_two_d_a[y_idx][x_idx];
    }
}
```

Here's a visual representation of the pointers and allocated blocks:

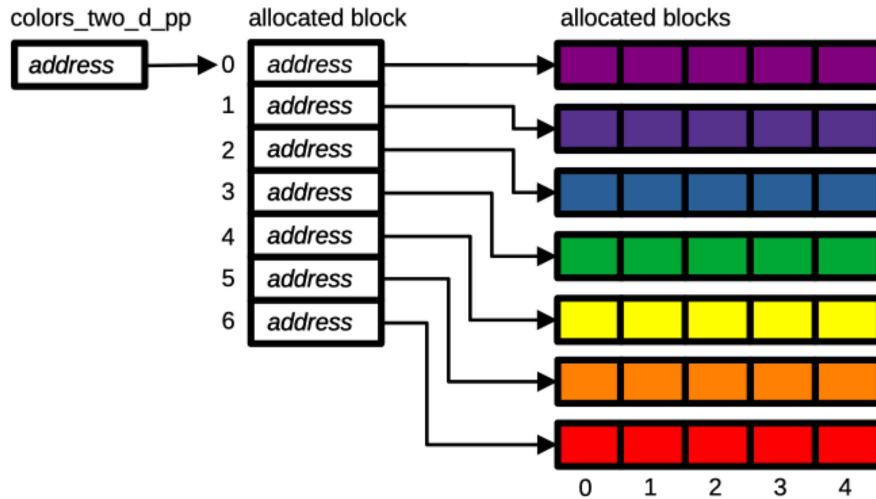

Figure 7: "Allocated Arrays of Pointers and Elements"

The key is to make sure the pointer types are correct, as the pointed-to type determines the stride. The first allocated memory block is cast to a pointer-to-pointer to color. When we apply the square brackets to set the pointer values to the allocated blocks of colors, in this line:

```
colors_two_d_pp[y_idx] =
    (color_t *)malloc( sizeof( colors_one_d_a_t ) );
```

the stride length used in the pointer math is the size of a pointer. This is equivalent to:

```
*( colors_two_d_pp + y_idx ) =
    (color_t *)malloc( sizeof( colors_one_d_a_t ) );
```

The line that sets the color values themselves then uses two sets of square brackets, applying pointer math consecutively:

```
colors_two_d_pp[y_idx][x_idx] =
    colors_two_d_a[y_idx][x_idx];
```

This uses two stride lengths, and is equivalent to:

```
*( *( colors_two_d_pp + y_idx ) + x_idx ) =
    colors_two_d_a[y_idx][x_idx];
```

This scales up to dimensions greater than 2, but note that the overhead for two-dimensional arrays of pointers will start to become significant, so I can't really recommend it. It is nice to be able to use the syntactic sugar of array notation for operations on both sources and destinations, though!

You can even do this all in one block of memory, with internal pointers:

```
void * colors_alloc_p =
    malloc( DIM_Y * sizeof ( color_t * ) +
            DIM_Y * DIM_X * sizeof( color_t ) );

color_t ** row_ptr_a_p = (color_t **)colors_alloc_p;
color_t * colors_a_p = (color_t *)( colors_alloc_p +
    DIM_Y * sizeof ( color_t * ) );

for ( unsigned int y_idx = 0; y_idx < DIM_Y; y_idx++ )
{
    row_ptr_a_p[y_idx] = colors_a_p + y_idx * DIM_X;
    for ( unsigned int x_idx = 0; x_idx < DIM_X; x_idx++ )
    {
        row_ptr_a_p[y_idx][x_idx] =
            colors_two_d_a[y_idx][x_idx];
    }
}
```

In the above code we allocate a single block of memory with **malloc()**, then create two pointers of more useful types that we can use to step through parts of this memory as arrays. The first part is treated as a block of 7 pointers to rows of color data. In this way we can perform a single allocation and still use the square bracket syntactic sugar, using explicit pointers instead of pointers that came from the degeneration of array types.

Note that if you want to play with the source code I used to write this part of the article, you can find it on GitHub here.

If you are a C programmer who did not have a deep understanding of how C array types really work under the hood, I hope this article has been useful. If you got all the way to the end, and tried running the code, and stepped through it using a debugger, and a light bulb came on for you — congratulations! You now understand a lot about C's "degenerate" array types than most C programmers do. Happy coding!

As always, this content is available for your use under a Creative Commons Attribution-NonCommercial 4.0 International License. If you'd like to help feed my coffee habit, you can leave me a tip via PayPal. Thanks!

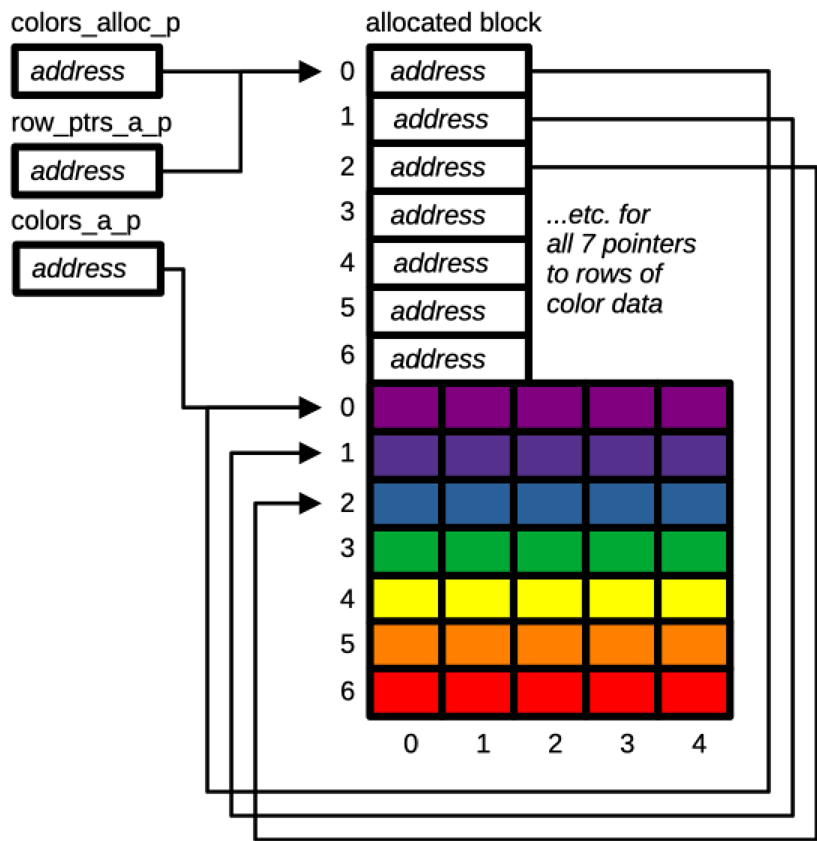

Figure 8: “A Single Allocation with Internal Pointers”