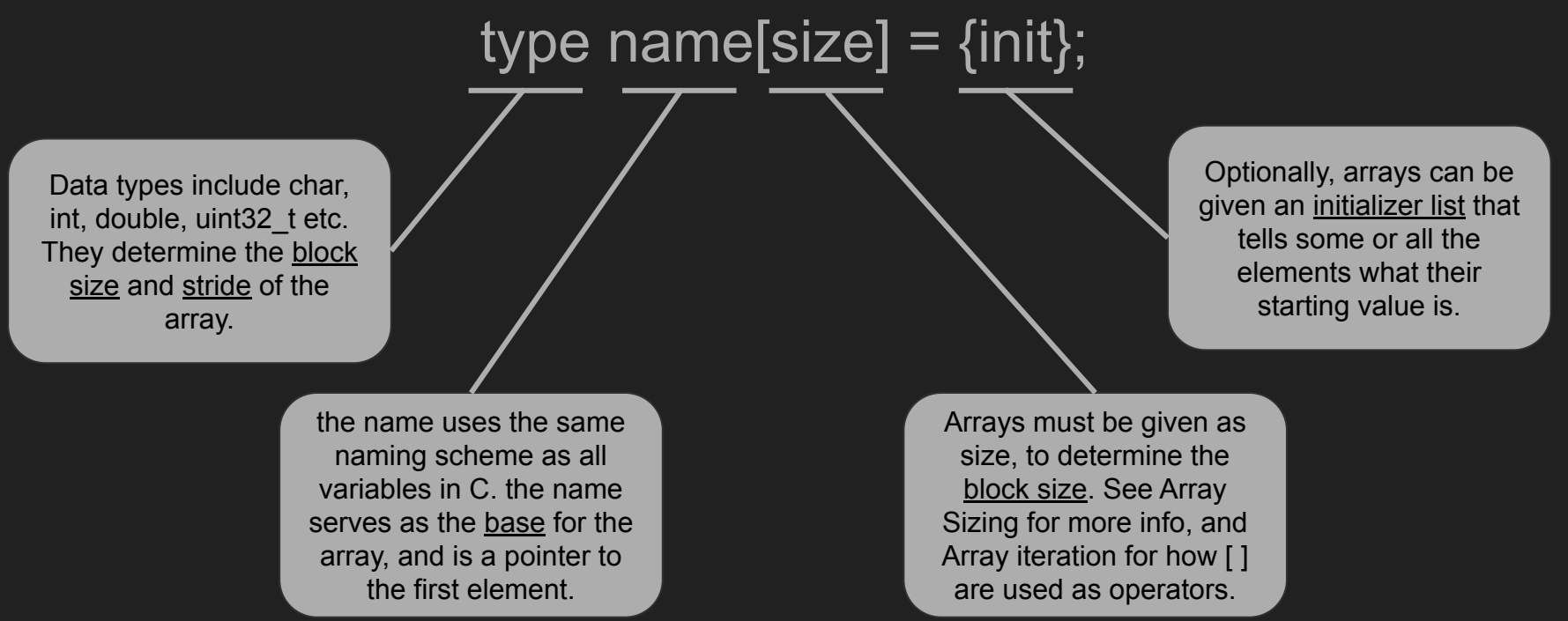
Working with arrays, multidimensional arrays in C

Array Declaration in C:



Array Initialization

Whenever you declare a variable in C, it has to be ‘initialized,’ so that it can be used in a program. How an array is initialized depends on the scope of its declaration, and whether or not you give it values.

When you declare an array in C, one of three things can happen:

1. You give it an initializer list for some or all the elements.

```
int arrayone[3] = {1,2,3}          // initializes to {1,2,3} because you gave it values.
```

2. If no values are given, then the initialization depends on the scope:
   a. If the array is *global* or *static* it is initialized to zero:

```
static char arraytwo[5];       // initializes as {0,0,0,0,0}
```

   b. If the array is within a *local* scope, it simply takes a block of memory, but does not set all the bits to zero. This means you don’t know what’s in those values when you use them, unless you set them after declaration but before using them.

```
int arraythree[4];             // initializes as ‘garbage’ - you don’t know what it is
```

Array Block Sizing

Arrays take up space. How do we know how much space they take up? Two things determine that: the data type given, and the size of the array!

Remember:
G means, 'garbage,' or that you cannot know what is stored in that value, but also you cannot assume its 0.

Arrays need a size! There are three valid ways to give an array a size:

1. With a size
    a. `int array[5];` // takes up 20 bytes; initializes to G,G,G,G,G
2. With an initialization matrix
    a. `short other[] = {1,2,3,4,5,6,7,8}` // takes up 16 bytes; initializes to 1,2,3,4,5,6,7,8
3. With both!
    a. `char thingone[5] = {1,2,3,4,5}` // takes up 5 bytes; initializes to 1,2,3,4,5
    b. `int thingtwo[5] = {6,7,8}` // takes up 5 bytes; initializes to 1,2,3,G,G

Array Iteration

Whenever you use an array, you are doing simple operations on it. Arrays know what value you are looking for because of the base given, the stride, and the index operator given.

remember that arrays are 0 indexed! this means array[5] has 5 elements labeled 0-4. The largest element idx is always *size - 1*

Base:

The base of an array is its name, which acts as a point to the first byte of memory the array controls. However, unlike a pointer, an array base cannot be reassigned (otherwise the array would be lost).

Stride:

The stride of an array is the distance, in units of bytes, that an array pointer moves as it indexes through an array. The stride matches the size of the data type given, so that the array indexes cleanly through itself. (i.e char = 1, int = 4, etc.)

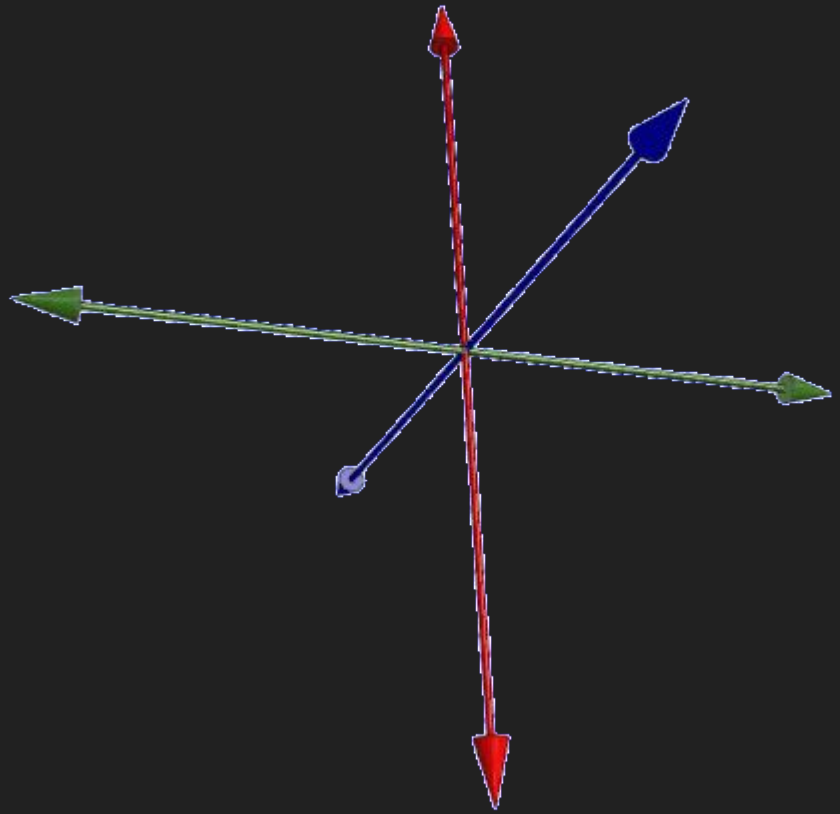
Index Operator:

The '[x]' is called an index operator, and actually isn't specific to arrays. When you use the index operator (except for in declaration), you are in essence saying, "start at the base, stride x times and return that result."

Examples:

```
int array[10] = {0,1,2,3,4,5,6,7,8,9};      // in this case [ ] is not an index operator, but part of the declaration
printf("%d\n", array[5]);                   // "5" - start at the base (which points to the 0), and stride 5 units
printf("%d\n", array[10]);                  // "G" - this isn't illegal, though it is out of bounds. Be careful with indexing!
```

Multi-dimensional Arrays



Rules of Multi-dimensional Arrays

All the dimensions of the array must be specified at declaration. You may omit the size of the first dimension (left to right) if you provide an initializer list.

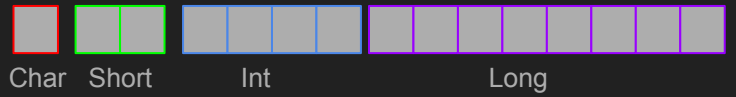
```
int array[2][4][6];                          // creates a list of size 2 x 4 x 6 with all garbage values.
char second[ ][3] = { {1,2,3}, {4,5,6}, {7,8,9} };   // creates an array of size 3 x 3 initialized with that list.
short another[ ][4] = {1,2,3,4};             // creates an array of size 1 x 4 initialized to 1,2,3,4.
```

Multi-dimensional arrays still only use one data type, which means that the stride works the same as before, as does the base.

Arrays are stored in row-major order, which means that they are stored in contiguous memory the same way that single dimension arrays are.

Why use them then, if they are just weirdly notated single dimension arrays? Because it allows us as humans to do have a much more visual interface for interacting with large quantities of data.

Visualizing Multi-dimensional Arrays

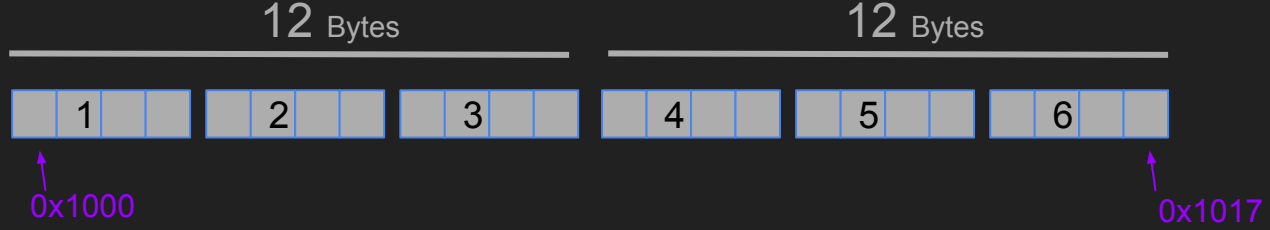


int runners_times[2][3];

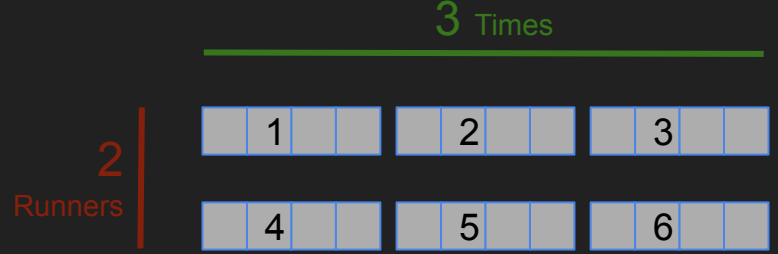
An array that has the time in seconds two different runners ran in three different races.

Notice how much easier it is to consider the array as “two dimensions” and let the computer decide where to literally put the data. Memory visualization is nice to know how much space the array takes up, but the data visualization is how we as humans will likely interact with our arrays.

Memory Visualization:



Data Visualization:

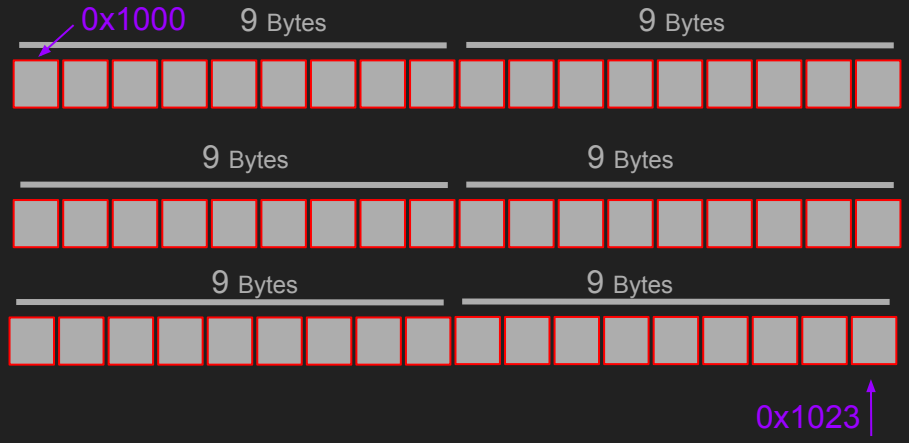


Visualizing Multi-dimensional Arrays

char rubix_cube_state[6][3][3];

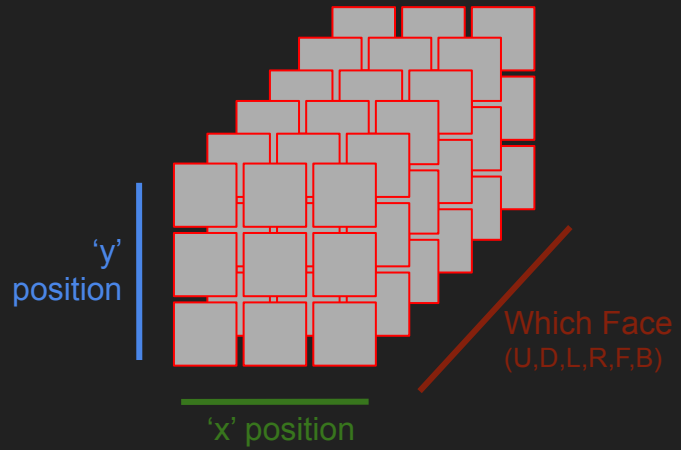
An array that store the state of a rubix cube by storing the color of each square at each location.

Memory Visualization:

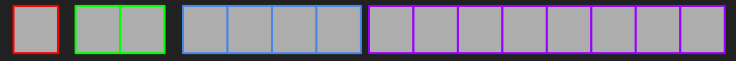


When using memory visualization, we have no idea where the data we want is located in the array. With data visualization, we have no idea where in memory we would find the data, though we can see it. This duality gets compounded the bigger the array, and we quickly lose the ability to the position of the data, and analyze the data at the same time.

Data Visualization:



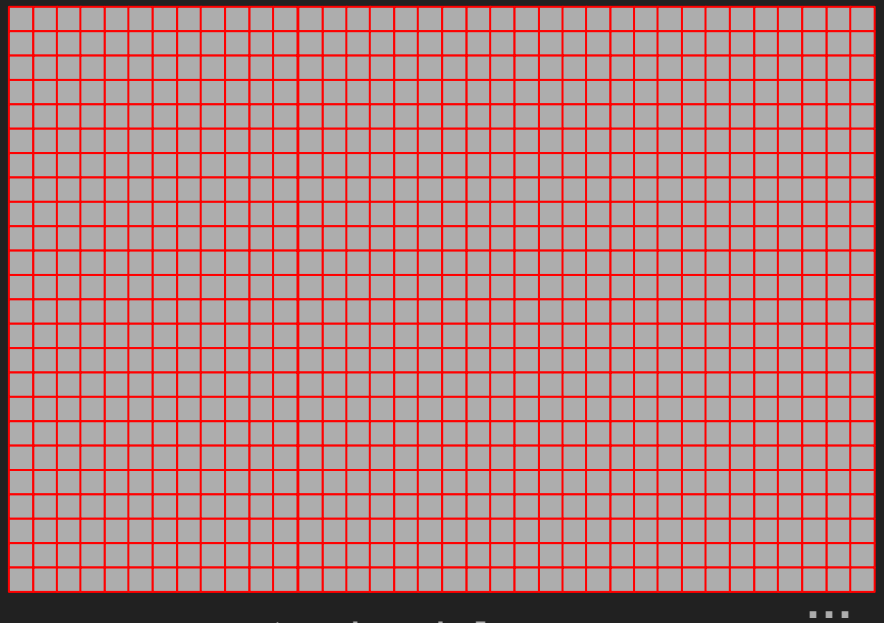
Visualizing Multi-dimensional Arrays



char gif_data[100][1080][1920][3];

An enormous array that store a 100 frame gif by pixel and then by the RBG value of that pixel.

Memory Visualization:



too hard :[

Data Visualization: