

## Arrays of Arrays

- **Objectives**
  - Consolidate understanding of one dimensional arrays
  - Extend understanding to multi-dimensional arrays
- **Contents**
  - Declaring arrays of arrays
  - Dangers of nested [ ] notation
  - Treating arrays of arrays as linear
  - Pointers to arrays
  - Passing arrays of arrays to functions
- **Practical**
  - More tests, more practice
- **Summary**



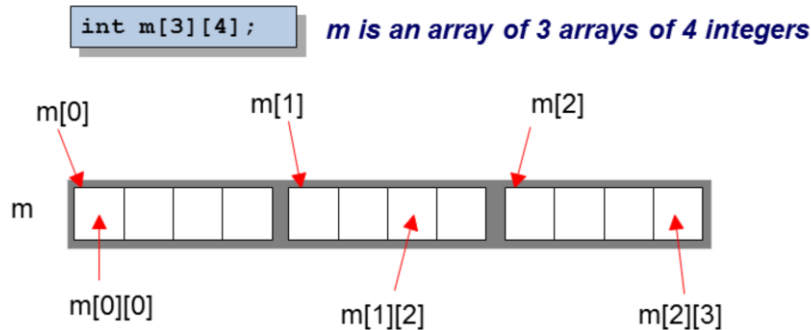
In this chapter, we shall investigate one of the most confusing aspects of C, namely the way in which arrays of arrays are declared and manipulated. We begin by looking at how C compilers lay out elements in arrays of arrays and we discuss the inherent inefficiency of the [ ] notation in these cases.

We shall then look at a variety of alternative schemes using simple pointer arithmetic, outlining the most common programming mistakes as we do so. The use of pointers to arrays will be discussed and the advantages and difficulties of this approach will be highlighted.

The chapter closes with a look at how arrays of arrays may be passed into functions, using both [ ] notation and pointer notation.

## Declaring arrays of arrays

- **C does not support multi-dimensional arrays**
- **However, it is possible to declare an array of any type, including an array of arrays**



It is a commonly-held misconception that C supports multi-dimensional arrays. It does not.

However, in C it is possible to declare an array of any type. Arrays of `ints`, arrays of `doubles`, arrays of structures, even arrays of arrays.

The array of arrays is the closest C comes to handling multi-dimensional arrays. The concept is quite straightforward: each element of the array is another array. A subscript is necessary to index the array, giving a second array. The second array requires a second subscript. If a third array were found, this would require a third subscript, and so on.

With arrays of arrays, the single most important thing is to read the declaration properly. For instance, consider the following:

```
double a[4][9][8];
```

You will get nowhere by saying "a is a multi-dimensional array of doubles, 4 by 9 by 8". Reading the declaration properly gives us:

"a is an array of 4 arrays of 9 arrays of 8 doubles".

With the declaration properly read and understood we can build up a picture of the nature of this beast.

## Initialising arrays of arrays

- A proper understanding of the declaration makes initialisation straightforward

- Note that the braces are optional

```
int a[3][4] =
{
    { 10, 11, 12, 13 },
    { 14, 15, 16, 17 },
    { 18, 19, 20, 21 }
};
```

*a is an array of 3  
arrays of 4 Integers*

```
double f[2][3][4] =
{
    {
        { 1.0, 2.0, 3.0, 4.0 },
        { 4.0, 5.0, 6.0, 7.0 },
        { 2.0, 4.0, 6.0, 8.0 }
    },
    {
        { 1.1, 2.1, 3.1, 4.1 },
        { 4.1, 5.1, 6.1, 7.1 },
        { 2.1, 4.1, 6.1, 8.1 }
    }
};
```

*f is an array of 2  
arrays of 3 arrays  
of 4 doubles*

Providing the declaration of arrays of arrays is fully understood, initialising it should be no problem. Consider the following simple example:

```
int a[9][5];
```

This declares `a` as an array of 9 arrays of 5 integers. Thus, we know that there are going to be 9 items. Each of these items is an array of 5 integers. Since arrays are always initialised using sets of `{ }` braces, we would expect there to be 9 sets of `{ }`, each one containing 5 integers.

Consider the following more complex example:

```
long double b[3][5][2][10];
```

This declares `b` as an array of 3 arrays of 5 arrays of 2 arrays of 10 long doubles. Thus, we know that `b` is an array of 3 things. So somewhere there must be 3 sets of `{ }` pairs. Within each of these 3 pairs of braces there must be 5 other pairs of `{ }` to initialise the 5 arrays held there. Within each of these 5 pairs of braces there must be 2 further pairs of `{ }` to initialise both arrays. Finally, within the last set of `{ }` come the 10 long double values we wish to hold.

## Letting initialisations default

- With arrays of arrays, only the first dimension may be omitted. This ensures that each sub-array is a fixed size
  - The compiler can count the number of items in an array

```
int a[] = { 5,4,3,2,1 };
```

```
int m[][5] =
{
    { 9, 8, 7, 6, 5 },
    { 6, 5, 4, 3, 2 },
    { 6, 5, 4, 5, 6 },
    { 9, 8, 7, 8, 9 }
};
```

```
double d[][2][3] =
{
    {
        { 1.0, 2.0, 3.0 },
        { 4.0, 5.0, 6.0 }
    },
    {
        { 1.1, 2.1, 3.1 },
        { 4.1, 5.1, 6.1 }
    },
    {
        { 1.2, 2.2, 3.2 },
        { 4.2, 5.2, 6.2 }
    }
};
```

When initialising an array, the compiler can safely be left to count up the number of elements specified and allocate the correct amount of storage.

Note that although it IS syntactically valid to declare an array without a size or initialising sequence, no space is allocated for an array. The statement

```
int d[];
```

is valid but does not define an array. It serves as a tentative definition if defined outside a function. If it is defined inside a function body, it is an error. It does, however, serve as a pointer to `int` if it is a parameter.

This is because the compiler must know how much storage to allocate to `d`. The only way it can know this is if you specify the dimension or if it can count up the number of elements specified in the initialisation.

Note also that the arrays in the slide shown above are *not* dynamic arrays. The size is fixed when the program is compiled and also when it runs. There is no support within the compiler for dynamic arrays.

With arrays of arrays, the first dimension may be omitted, but all other dimensions must be specified. In the declaration of the array `m` shown above, the compiler can check that arrays of 5 integers are used to initialise each of the rows in the array.

In the initialisation of the array `d`, the compiler checks that 2 arrays of 3 doubles are provided for each of `d[0]`, `d[1]` and `d[2]`.

At the end of the day, it is only a consistency check. If all programmers could be guaranteed always to initialise their arrays correctly, this restriction would not be necessary. However, inevitably, human nature dictates that it is safer to have this feature than to omit it.

## Initialisations

- Only the *first* dimension may be omitted to reduce the number of permutations and combinations:

```
double garbage[][][] =  
{  
    1.1, 2.2, 3.3, 4.4, 5.5,  
    6.6, 7.7, 8.8, 9.9,  
    7.7, 8.8, 9.9, 10.1, 13.0, 15.0, 7.8,  
    8.2, 9.3,  
    9.1, 4.5, 3.7, 8.9, 5.3, 4.3  
};
```



- Are these 24 doubles to be arranged as...  
2 lots of 2 lots of 6, or  
3 lots of 4 lots of 2, or  
2 lots of 3 lots of 4, or .....?



This slide illustrates the sort of nonsense the compiler would have to put up with if it did not enforce the rules already stated.

Clearly the compiler can count the number of elements to arrive at 24. The question is how should those 24 elements be distributed within the three dimensions of the array?

## The [ ] notation with arrays of arrays

- The overhead of using [ ] notation with arrays of arrays can be significant:

```
int    a[5][7][9];
size_t i, j, k;

for (i = 0; i < 5; i++)
    for (j = 0; j < 7; j++)
        for (k = 0; k < 9; k++)
            a[i][j][k] = 0;
```



*This statement executes  
 $5 \times 7 \times 9 = 315$   
times*

- The problem is not only the number of times the statement is executed, but also the 3 multiplications that need to be done
- It can be very important to enable the optimiser

We have already seen that there may be problems with the code the compiler produces when converting from array notation to pointer and offset notation. Unless optimised away, a multiplication will be involved.

When arrays have large numbers of elements, and each element access requires the multiplication, any inefficiencies can scale up significantly.

It has to be said that if compilers can optimise away the one multiplication involved in accessing the elements of a single dimensional array they will also be able to optimise away two, three or more multiplications.

### Use of nested [ ] - conclusions

- With our array of arrays of arrays, each access to an element of the array requires three multiplications to be performed
- Clearly, there will be  $n$  multiplications required for each access of an  $n$ -dimensional array
- A good optimiser is needed



The situation is bleak. The implication is that if we need "multi-dimensional" arrays (arrays of arrays), the compiler is going to generate horribly inefficient code. Previously, the situation seemed to improve when the array notation was dispensed with, but is this so straightforward with arrays of arrays?



## "Cheating" with arrays of arrays

- Since the compiler allocates arrays in contiguous storage. A pointer to the "base" type can be used to access all the elements of the array:

```
int a[3][4][5];
int * p;
int * end;

for (p = &a[0][0][0], end = &a[2][3][4]; p <= end; p++)
    *p = 0;
for (p = &a[0][0][0], end = &a[3][0][0]; p < end; p++)
    *p = 0;
for (p = &a[0][0][0], end = &a[2][3][5]; p < end; p++)
    *p = 0;
for (p = &a[0][0][0], end = &a[3][4][5]; p < end; p++)
    *p = 0;
```



Arrays are always stored in contiguous memory. This is independent of the type of the array (even though we may be dealing with an array of arrays).

The easiest way to "cheat" is to pretend that our "multi-dimensional" array is really single-dimensional. We can set a pointer to the start of the array and keep incrementing it, checking that the pointer has not moved outside the array.

One problem here is in determining which element is the last element of the array. It is always important to remember that arrays start from 0 in C. This is just as true of arrays of arrays.

Thus in the example, for an array of 3 arrays of 4 arrays of 5 integers the last element will be at location  $[3 - 1 = 2]$ ,  $[4 - 1 = 3]$ ,  $[5 - 1 = 4]$ , i.e. location  $a[2][3][4]$ .

To calculate the location of one element beyond the end of the array requires a little knowledge of how the compiler counts. Clearly, the location  $a[2][3][5]$  will be one beyond location  $a[2][3][4]$ . Since we have already established that  $a[2][3][4]$  is the last valid element,  $a[2][3][5]$  must be the first invalid element beyond the end of the array.

Another way of addressing this first invalid element is  $[3][0][0]$ .

It is important to see why the element  $a[3][4][5]$  is a bad one to choose if you are trying to address the first invalid element. This will address an element which is **25** locations beyond the end of the array.

We admit that none of the above are particularly readable. For each one we have lost the clarity of the original code on the previous page. Use of multiple `[ ]` may be inefficient (without an optimiser) but they are certainly clearer.

## Pointers to arrays

- C defines a "pointer to an array" type which does not behave in quite the same way as a "base" pointer

```
int * m;           /* m is a pointer to an int */

int (*p) [5];      /* p is a pointer to an array of 5 ints */

int (*q) [4] [5];  /* q is a pointer to an array of */
                  /* 4 arrays of 5 ints */
```

- When `m` is incremented it moves over 1 integer; when `p` is incremented it moves over 5 integers; when `q` is incremented it moves over 20 (4 x 5) integers
- Although simple in concept, pointers to arrays are not quite so straightforward to use

In order to help to solve some of the problems already raised (and to create a few more), C defines another kind of pointer. C supports the notion of a "pointer to an array" or even a "pointer to an array of arrays", and so on. The first thing to realise about these pointers is the syntax required in the declaration, which is very different from the syntax of "ordinary" pointer declarations.

Note the brackets within the declaration. Without these brackets, `p` would be declared as "an array of 5 pointers to `int`":

```
int *p[5];
```

Now consider the declaration of `p` with the correct use of parentheses:

```
int (*p) [5];
```

With the brackets in their proper position, `p` is a pointer to an array of 5 `ints`. The `*` operator is bound to `p` by the parentheses, even though the `[]` operator has a higher precedence. *Only one pointer is declared*, not an array of them. Consider the following example:

```
long double (*z) [9] [3] [7];
```

`z` is a pointer to an array of 9 arrays of 3 arrays of 7 `long doubles`. Again, there is only one pointer, not an array of them.

The difference between pointers to arrays and other pointers is the amount by which they are scaled when they are incremented. For `m`, declared above, incrementing by 1 scales the pointer by `sizeof(int)`. For `p`, declared above, incrementing by 1 scales the pointer by 5 times `sizeof(int)`. On a typical 32-bit machine this would be 20 bytes added to the pointer every time it is incremented. When the variable `q` is incremented by 1, a scaling of 20 times `sizeof(int)` takes place. This ability to have a large number added to a pointer every time it is incremented makes pointers very useful when dealing with arrays of arrays.

## Initialising pointers to arrays

- When initialising a pointer to an array an "&" must be placed before the array name
- This is unusual since the name of an array is already an address

```
int a[8] =
{
    19, 20, 21, 22,
    23, 24, 25, 26
};
int (*pa)[8] = &a;
```

```
double b[2][3] =
{
    { 0.0, 0.1, 0.2 },
    { 1.0, 1.1, 1.2 }
};
double (*pb)[2][3] = &b;
```

```
long c[2][3][2] =
{
    {
        { 0, 1 },
        { 0, 1 },
        { 0, 1 }
    },
    {
        { 5, 6 },
        { 5, 6 },
        { 5, 6 }
    }
};
long (*pc)[2][3][2] = &c;
```

Pointers to arrays must be initialised in a rather unusual way:

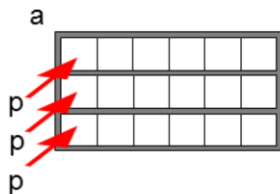
```
int (*pa)[8] = &a;
```

It is rather odd to see an & used with the name of an array, since the name of an array yields the address of its first element. How can we take the address of an address?

The initialisation makes sense if you consider the "level" of the pointer `pa`. It is a level 2 variable since both `*` and `[ ]` are used. The level of `a` is 1, since only `[ ]` appear in the declaration. An `&` is needed to raise the level of `a` from 1 to 2 so that it may be assigned.

## Using pointers to arrays

- A "base" pointer may be used to move along rows
- A pointer to an array may be used to move up and down columns



```
int a[3][6] =
{
    { 1, 2, 3, 4, 5, 6 },
    { 7, 8, 9, 10, 11, 12 },
    { 13, 14, 15, 16, 17, 18 }
};
int (*p)[6] = &a[0];
int (*end)[6] = &a[3];
for ( ; p < end; p++)
{
    int * q = *p;
    printf("%d\n", *q);
}
```

1  
7  
13

When a pointer to an integer is incremented it moves by one integer. By initialising an integer pointer to the first of the arrays of arrays would allow access to each of the 6 integers stored there.

When a pointer to an array of 6 integers is incremented it moves by 6 integers. Initialising a pointer to an array of 6 integers to point to the first of the arrays of arrays would allow access to the integers in the first location of each array.

The array pointer can be thought of as accessing the array on a column basis.

## Exercise: what does this print?

```
int a[3][6] =  
{  
    { 1, 2, 3, 4, 5, 6 },  
    { 7, 8, 9, 10, 11, 12 },  
    { 13, 14, 15, 16, 17, 18 }  
};  
int (*p)[6] = &a[0];  
int (*end)[6] = &a[3];  
for ( ; p < end; p++)  
{  
    int * q = *p;  
    printf("%d\n", q[3]);  
}
```




## Arrays of arrays as parameters

- When an array is passed to a function, the parameter may be declared as a pointer
- When an array of arrays is passed to a function, the parameter may not be declared as a pointer to a pointer:

```
void f(int *);

int main(void)
{
    int a[10];
    f(a);
    return 0;
}


void f(int * p)
{
    p[2] = 99;
}
```



```
void f(int **);

int main(void)
{
    int a[5][10];
    f(a);
    return 0;
}

void f(int ** p)
{
    p[2][3] = 99;
}
```



As seen in the previous chapter, when an array is passed down to a function the parameter to the function may be declared as a pointer.

Thus it would seem logical that when passing an array of arrays down to a function the parameter should be declared as a pointer to a pointer. However, this is not the case.

In the example above left, in the function `f` the access `p[2]`, which is equivalent to `*(p + 2)`, causes the compiler to move two integers beyond where `p` points and place the 99 in that location.

In the example above right, the access `p[2][3]` would be equivalent to `*(*(p + 2) + 3)`. With `p` declared as a pointer to a pointer, `p+2` will move forward by two pointers (`2 * sizeof(int *)` - 8 bytes on a typical 32-bit machine). Yet this is not correct. To access the element we'd *like* to access, `p+2` should move over the first two blocks of 10 integers. This would involve a jump of `2 * 10 * sizeof(int)` bytes (2 blocks of 10 integers).

## Declaring parameters correctly

- When passing an array of arrays to a function the pointer needs to "know" how much to jump by
- A pointer to an array is required

```
int a[5][10];  
func4(a);
```

```
void func4(int (*p)[10])  
{  
    p[2][3] = 99;  
}
```

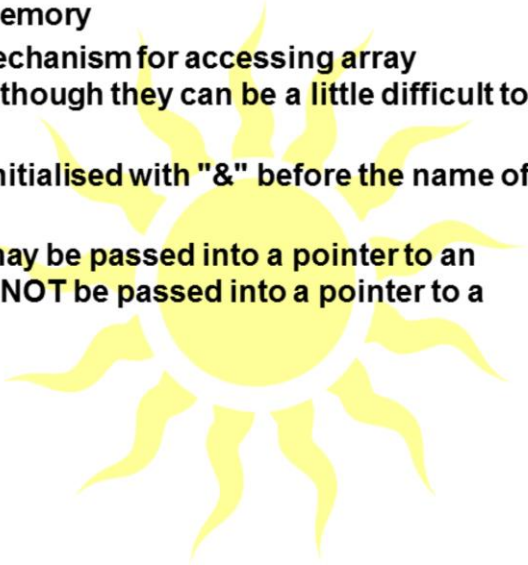
*the compiler knows to  
scale the 2 by the sizeof  
10 ints*

```
void func4(int p[][10])  
{  
    p[2][1] = 19;  
}
```

*this declaration of "p" is  
effectively equivalent to  
int (\*p)[10]*

Declaring the parameter as a pointer to an array fixes the problems previously seen. Now the compiler is able to scale any increment to be that of the appropriate size, that of the size of the array that the pointer points to. In our example, the calling function has an array of arrays which has 10 columns. A pointer to an array of 10 ints, i.e. an `int (*) [10]`, will be the appropriate type for traversing the array. In the called function, the compiler will be able to size the appropriate step when accessing the array, either by indexing, using the `[]` as shown above, or through appropriate `++` operation on the pointer itself.

### Summary

- **Arrays of arrays (etc.) may be treated as linear arrays since they are allocated in a single piece of memory**
  - **Pointers to arrays provide a mechanism for accessing array elements on a column basis, although they can be a little difficult to use**
  - **A pointer to an array must be initialised with "&" before the name of the array**
  - **Whereas an array of integers may be passed into a pointer to an integer, an array of arrays may NOT be passed into a pointer to a pointer to an integer**
- 

---

In this chapter, we have studied the use of arrays of arrays in C and have explored a number of alternative techniques for manipulating them in an efficient manner.

Pointers to arrays may be declared, although there are some difficulties, as we have noted during the chapter. In other situations, it might be more natural to treat an array of arrays as if it were a single linear array, and traverse the whole array using a simple "one level of indirection" pointer.

When initialising arrays of arrays, or indeed when passing them into a function, the left-most dimension specifier may be omitted. All other dimensions must be specified, however, so that the compiler knows the length of each "row" in the array.