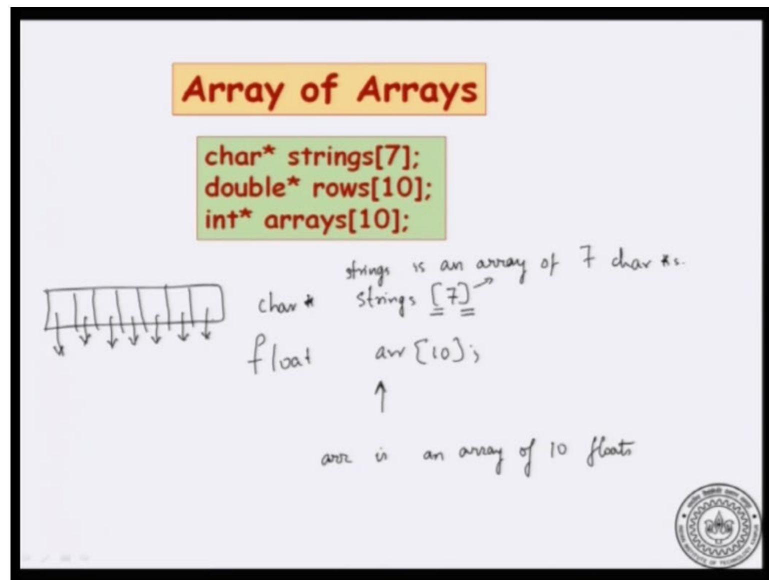
Introduction to Programming in C
Department of Computer Science and Engineering

In this video will look at the last possibility with respective multi-dimensional arrays in pointers, this is known as an array of arrays.

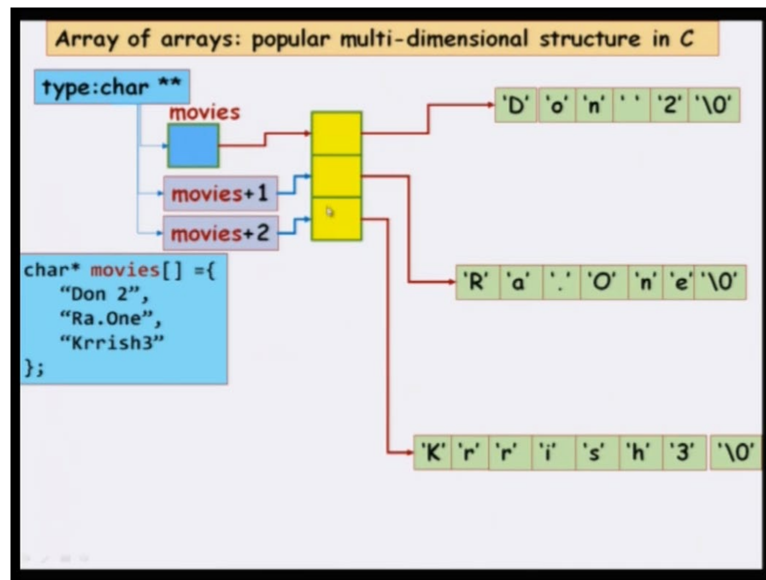
(Refer Slide Time: 00:04)



In order to understand this, let us just look at something we are comfortable with, if I had just a float arr[10] elements, then how would I read this, I will say that array is arr is an array of 10 floats. So, this is how I would read it, if I have more complicated declaration like char* as strings. So, notice that the precedence for this [ ], is higher than that of the precedence for *.

So, this would actually be read as strings is an array of 7 character stars. So, that is how it could be read, because 7 would bind closer to strings. So, strings will become in array of size 7 and what type is it, it is char *. So, you replace float with char * and it is roughly the same phenomenal. So, the pictorially you can think of it like this. So, you have 7 cells in strings and each entry is a char *. So, each entry is a character pointer, you can think of it as a string, you can think of it is a character array whatever. So, here is the pictorial representation. Let us look at why we would need such as structure and what is the advantage of it? This is very popular structure almost as popular as two dimensional arrays themselves.

(Refer Slide Time: 02:14)

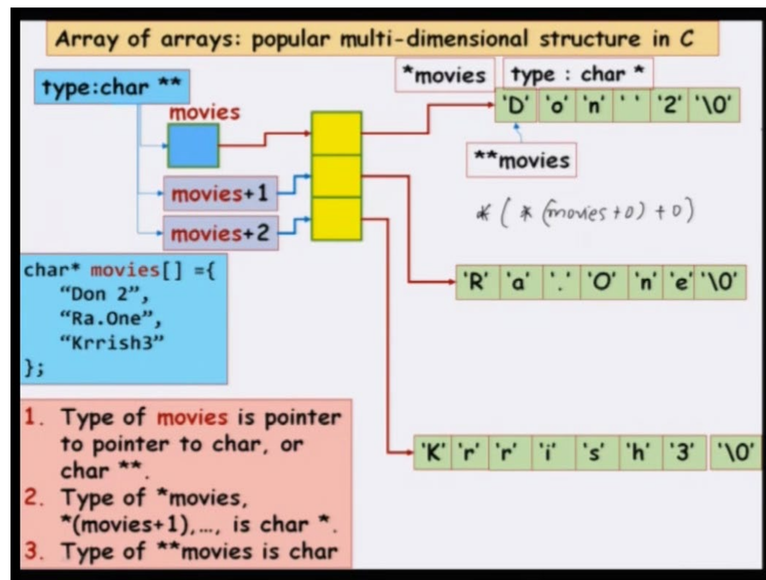


So, let us look at what it means for when we allocate character* an array of arrays. So, we may want to let say store the names of several movies. And one of the things is the there is no maximum limit to the name of a movie and it can be as long as we want it can be as short as you want and suppose you want to store all of these is in a data. So, let say that we have char *movies and I declare it as an array of arrays and it contains the first array is Don 2, the second is Ra.one and. So, on.

Now, how will we do this, So, one way to do this is you say that movies is pointing to an array of arrays. So, movies + 1 is pointing to another character array, movies + 2 is pointing to another character array and. So, on. So, this is how we pictorially represent it, there are three entries and each entry is a character pointer. So, it can point to any character array what. So, ever.

And here you see the distinct advantage of this kind of representation over 2D matrices. Why? Because, in 2D matrices the whole point was the number of columns was fixed, that is how the pointer arithmetic worked. Here, the number of columns in one row can be different from the number of columns in another row. So, this representation is actually more useful when you have what are known as ragged arrays, that is one row and the next row may have very different lengths. And here is natural situation of storing strings when you need such a facility.

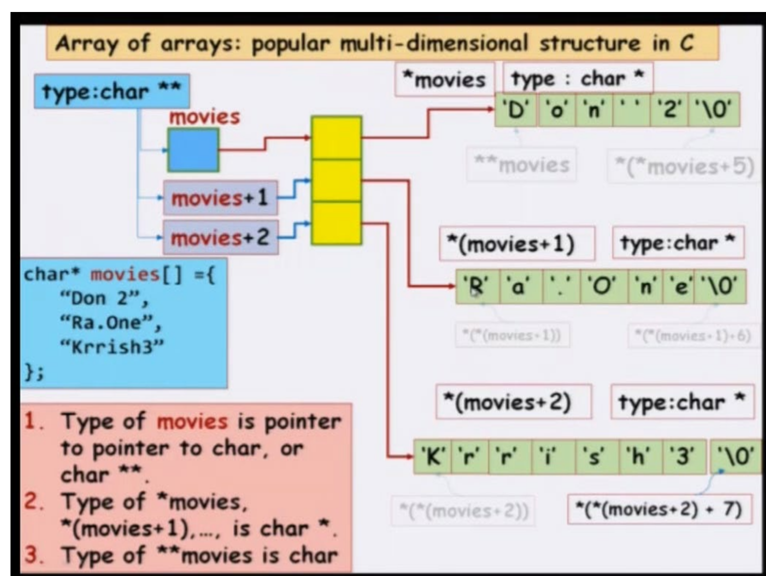
(Refer Slide Time: 04:08)



So, let us see what this means type of the variable movies is a pointer to a pointer to character or char**. Now, type of *movies is char*, because you dereference one level and type of **movies is char. So, let us look at it once more. So, *movies has type char *. So, in particular *movies will be this array, it is pointing to this array. So, **movies will be what is it according to the general formula, this will be *(*(movies plus 0) + 0). So, this will be the pointer arithmetic version of accessing this cell which contains D.

But, instead you could also write movies[0][0]. Similarly, in order to get to the last cell here, you could say *(*(movies + 5)), it is the particular application of the general formula.

(Refer Slide Time: 05:32)



The second row will be *(movies + 1) again try to think in which ever notion your comfortable with. Because, you can also right this as movies 1, you will get the same result. So, *(movies + 1) will come to the second row and you have *(*(movies + 1) that would come to the first element in the second row and. So, on. So, *(movies + 2) would be the third array in the structure and here is how you access different elements in the third array.

So, notice the picture is slightly different here, even the representation suggest that, these rows need not be contiguous in memory. So, then location after this row n's need not be this row. So, the second row can be located arbitrarily far away in memory, the advantage due to that is that these rows can be of different length, they are not packed as in the 2D array.

(Refer Slide Time: 06:54)

```
char *movies[] ={
    "Don 2",
   "Ra.One",
   "Krrish3",
};
```

movies
movies+1
'D' 'o' 'n' ' ' '2' '\0'
'R' 'a' '.' 'O' 'n' 'e' '\0'
'K' 'r' 'r' 'i' 's' 'h' '3' '\0'

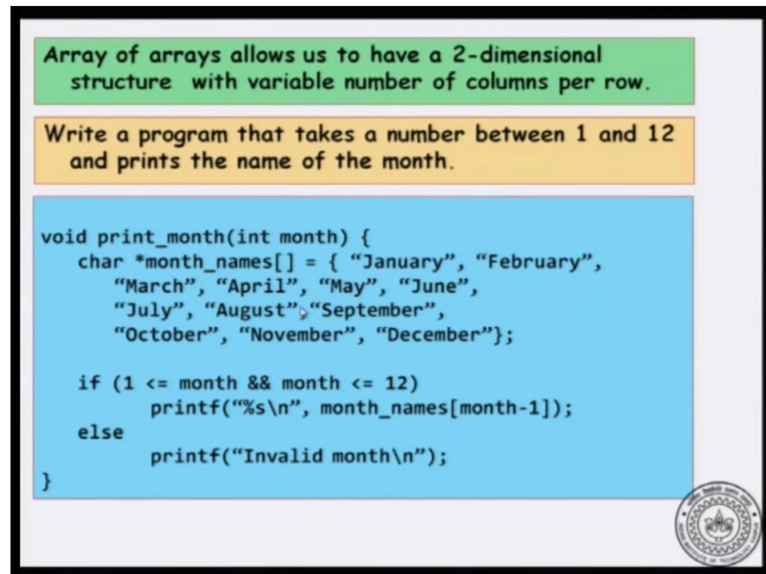
What are the outputs of?	Output
printf("%s", movies[2]);	Krrish3
printf("%s", movies[0]);	Don 2
printf("%s", *(movies+1));	Ra.One
putchar(*(*movies+1)+1));	a
putchar(*(*movies+2)+3));	z

Let us look at this particular thing in detail. So, that you get comfortable with in. So, suppose you have that array and I considered what is printf("%s", movies [2]), movies 2 will be the third character array, that is present in the structure. So, it will print Krrish 3. Similarly, movies 0 will print the first string and if you say printf("%s", *(movies + 1) by pointer notation, this is the same as the subscript notation movies[1]. So, that would print Ra.One.

Now, what happens if you have put char *(movies + 1 + 1). So, again if you are more comfortable with a subscript notation, you can translate back in to the subscript notation, this will become movies[1][1]. So, what it will print is, this letter which is small a,

similarly for the last one. So, it will print whatever it will print the i. So, here is a whereas...

(Refer Slide Time: 08:24)



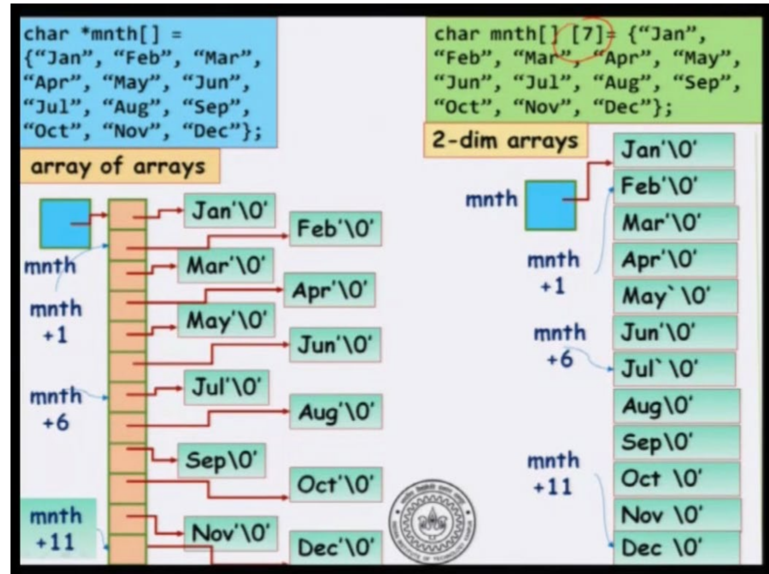
Array of array is now allows us to have a two dimensional structure with different number of elements per row and this is the advantage that it has. Now, let say that we want to right a very natural program, which is it takes a number between 1 and 12 and it prints out what is that month name corresponding to that number. So, I want to store in months and here is the problem different months have different lengths. We right now saw is solution to this problem, which is to store arbitrary length a strings in one structure, we would make an array of arrays.

So, you can say that char *month_names[ ]. So, this is an array of arrays of character and then you can just initialize it to the month names, you do this and then I will write the code. So, you can write the code in anyway. So, you can say that 0 is January and. So, on up to 11 is December. But, maybe it is more natural to say that 1 is January and. So, on up to 12 is December. So, I will check if the given month index is between 1 and 12, then I will print the month name month minus 1.

So, if you give the month as 1 you will print month names 0 which is January, if you give the month as 2 you will print month names 1 which is February and so, on. Now, if the month is not in this range it is in invalid month. So, you just print that an exist, So, here is a very simple program with illustrates what advantage you get out of this kind of array of array structure. You can store with in the same data structure different strings of

completely different lengths, this is not possible in a 2D array because, all you have to calculus something like the maximum column length. So, the maximum width month name for example, it could be September and then all the other names have to have exactly that width.

(Refer Slide Time: 10:48)



So, let us look at this the array of arrays picture is like this, you have an array of characters stars. Now, each of those characters stars may be pointing to different months. In this every month is exactly three characters long,, but you get the picture basically in this these rows can be of different lengths. So, contrast this with two dimensional arrays, where the chief feature of the two dimensional array is the following, you have to specify the number of columns.

So, the number of columns have be to specified and no matter what the exact string is, it will occupies 7 characters now. So, the remaining will be null fill or something. So, also notice that pictorially I have tried to represent it, the very next memory cell after the first row will be the beginning of the second row. So, after row 0 it will immediately start with row 1. Whereas, in the case of array of arrays row 0 and row 1 may be located arbitrarily for apart in memory.

The only connection is that the pointers to these rows are consecutively located in the pointer array, that is not the case here, it is actually located together in memory and it is represented in row major fashion, where each row will take exactly 7 letters. So, I hope the limitation of the two dimensional arrays in this case is clear.

(Refer Slide Time: 12:31)

So, you can try a few exercises in order to understand this notation a little bit, this concept of array of arrays a little bit better. So, let us look at the types of various concepts. So, if I have month, month is actually a char **, it is an address. So, if you want to print out month, I mean it is very rare that you need to print out month, you would use something like %l u, which is long unsigned for printing the unsigned long int. What happens if you access *month? Now, you dereferencing one level below. So, it will be a char *.

So, now, if you print percentages *month, it will print January. If you print a **month, you have dereference two levels and you will get the first letter of the first array, which is j and you can try out a few other thinks, you can say the *(month + 1). So, one way you can do it is you translated to subscript notation and try to see what it will print and there are... So, I would encourage you to try out these examples, in order to get that translation between arrays and pointers correct.

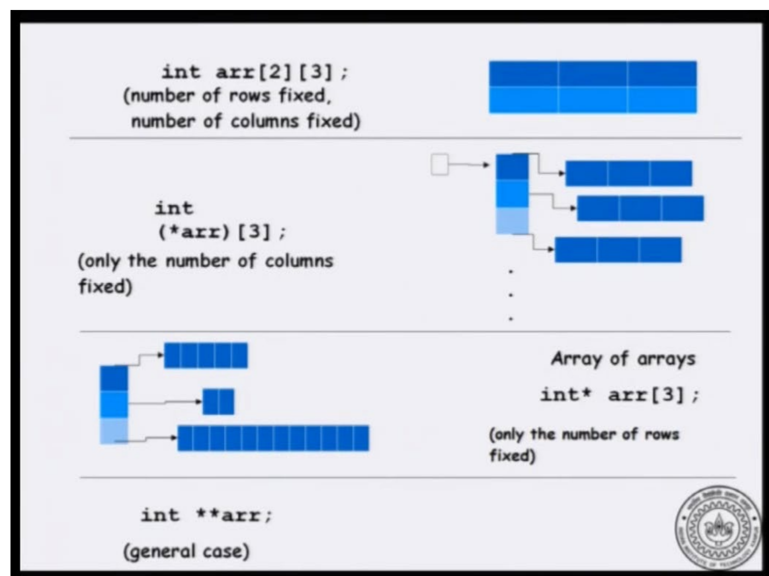
(Refer Slide Time: 14:01)

Comparison

Array of Arrays char *mnth[12];	2-dim arrays double matrix[10][10];
Individual array sizes can be different. E.g., mnth[6] is "July" (size 5). mnth[8] is "September" (size 10).	All rows must have same length.
Useful in string/word processing and representing graphs, non-uniform sized structures. Not so useful for matrices.	Natural for matrix computations. Not so good for non-uniform sized structure.

So, the comparison between array of arrays in two dimensional arrays on the one hand individual array sizes can be different in the case of array of arrays. In the case of two dimensional arrays all the rows must have exactly the same number of columns. So, array for array is useful in a lot of string processing routines, in representing graphs and something like that. But, two dimensional arrays are more advantages when you deal with matrices, because mathematical matrices typically have a fix number of columns.

(Refer Slide Time: 14:40)



So, here is wrong picture, but it short of gives you an idea of how to look at these structures. So, if I have int array[2][3] you can think of it as the number of rows is fixed and the number of columns is fixed, this is not actually what happens in C, in C actually

the number rows does not matter, the number columns matters. But, you can for a moment to make it easier to think about, think that if you declare it in this way, this is when the number of rows is fixed in the number of columns are fixed.

So, in particular if you know beforehand that you data structure has a fixed number of rows and fixed number of columns, then it is probably better to you said 2D array. Now, if you have int *arr[3], now this means that arr is a pointer to an array of size 3. So, here the number of columns is fixed. But, the number of rows is variable, it you can have any number of rows, on the other hand the third case int *arr[3]. So, it is an array of 3 elements each of type int *.

So, you can see that this is one situation, where you have 3 pointers,, but each of them can point to arrays of arbitrary length. So, this is a situation where the number of rows can be seen as fixed and the number of columns is variable. And the general case can be int **, which is where the number of rows and the number of columns are both waiting.

So, you can think of it and this way, this is not a correct picture. But, when you want to modal data, this is probably and you know that you are in a situation, where the number of columns is variable. But, the number of rows you know beforehand probably you should go for array of arrays. If you are in a situation, where you know that the number of columns is fixed, but you do not know how many data there are, then you can go for int* array of size 3. So, you can go for the second alternative and so, on. So, this picture is not quite accurate,, but it is indicative of the usage.