

**Introduction to Programming in C**  
**Prof. Satyadev Nandakumar**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kanpur**

**Lecture - 44**

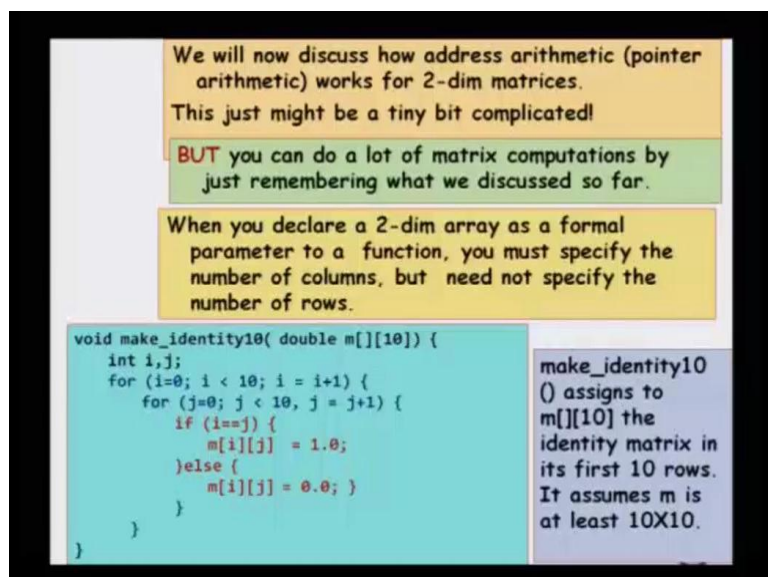
In this video, we will look at the relation between Multi-dimensional Arrays and Pointers.

(Refer Slide Time: 00:03)



And this is by far one of the trickiest topics in the entire course.

(Refer Slide Time: 00:18)

A slide with a light gray background and a black border. It contains several text boxes and a code block. The top text box (yellow) says: "We will now discuss how address arithmetic (pointer arithmetic) works for 2-dim matrices. This just might be a tiny bit complicated!". The middle text box (green) says: "BUT you can do a lot of matrix computations by just remembering what we discussed so far.". The bottom text box (yellow) says: "When you declare a 2-dim array as a formal parameter to a function, you must specify the number of columns, but need not specify the number of rows.". The code block (cyan) shows the function definition for make\_identity10. The bottom right text box (purple) explains that make\_identity10() assigns to m[][10] the identity matrix in its first 10 rows and assumes m is at least 10x10.

We will now discuss how address arithmetic (pointer arithmetic) works for 2-dim matrices. This just might be a tiny bit complicated!

**BUT** you can do a lot of matrix computations by just remembering what we discussed so far.

When you declare a 2-dim array as a formal parameter to a function, you must specify the number of columns, but need not specify the number of rows.

```
void make_identity10( double m[][10]) {
    int i,j;
    for (i=0; i < 10; i = i+1) {
        for (j=0; j < 10; j = j+1) {
            if (i==j) {
                m[i][j] = 1.0;
            }else {
                m[i][j] = 0.0; }
        }
    }
}
```

make\_identity10  
() assigns to  
m[][10] the  
identity matrix in  
its first 10 rows.  
It assumes m is  
at least 10X10.

And you can code multidimensional arrays without actually understanding the exact relation between multidimensional arrays and pointers. But, understanding this gives you a better grasp of how C treats multidimensional arrays. So, we will now discuss how pointer arithmetic works with two dimensional matrixes. Because, as soon as we had discussed one dimensional matrix, the next thing we did was we discussed the relation between pointers and 1D arrays.

So, let us try to see what is the relation between pointers and 2D arrays. Now, this is more complicated than it looks at first sight. And you can do a lot of matrix computations by not understanding this. Except that, understanding this gives you a better grasp for what is happening. We have seen that, when you declare a 2D array as a parameter to a function, then you should specify the number of columns, but not the number of rows.

So, let us look at a function which makes an identity matrix, an identity matrix is a matrix that has one along it is diagonal and zero every where else. So, we have void, make identity 10 and it takes a matrix of size double, the number of columns is 10. Since, identity matrixes are square matrixes, this essentially says that the code will work for a 10 by 10 matrixes.

Then, I have a for loop going from i equal to 0 to 10 and a for loop going from, for the columns going from j equal to 0 to 10. And the code just says that, if I am at a diagonal element that is i equal to j, then  $m[i][j]$  is 1 for all other elements,  $m[i][j]$  is 0. So, this creates a matrix of size 10 by 10.


(Refer Slide Time: 02:23)

Now suppose i want to make a 20 X 20 identity matrix. Do i have to write another function?

```
void make_identity20(double m[][20])
```

*But, i don't like complicated!*

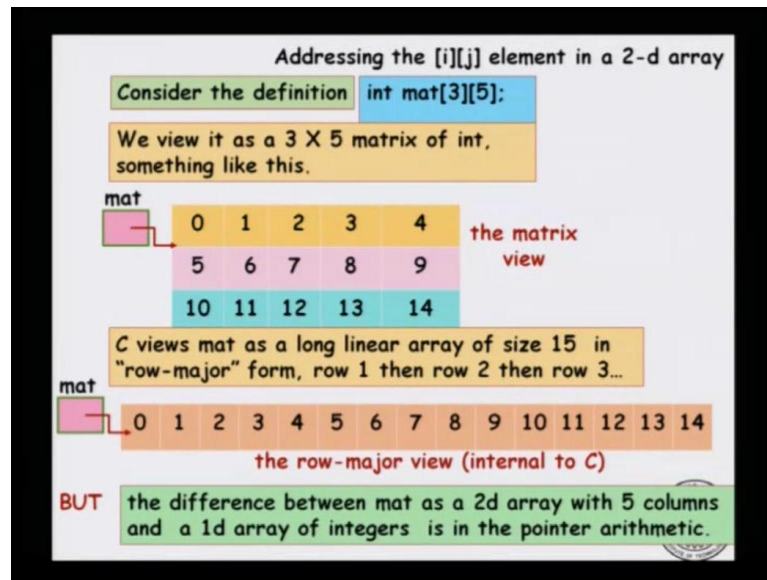
In C, that would be the standard way. There is a way around, but umm... it is a bit complicated. Let us now understand the address arithmetic for 2-dim arrays.



Now, this is a very strange code. Because, it is a function that essentially makes exactly one matrix. It would have been nice, if I would have a function that can create arbitrary size identity matrixes. For example, if I wanted a 20 by 20 matrix, it looks like, I have to write another function make identity 20 double m 20, the rows are unspecified, the number of columns is 20.

This is the standard way to do it. But, there is a slightly more complicated way to actually accomplish a function which can take an arbitrary size. So, let us see how these things can be done by understanding, how pointer arithmetic works with 2D arrays.

(Refer Slide Time: 03:15)

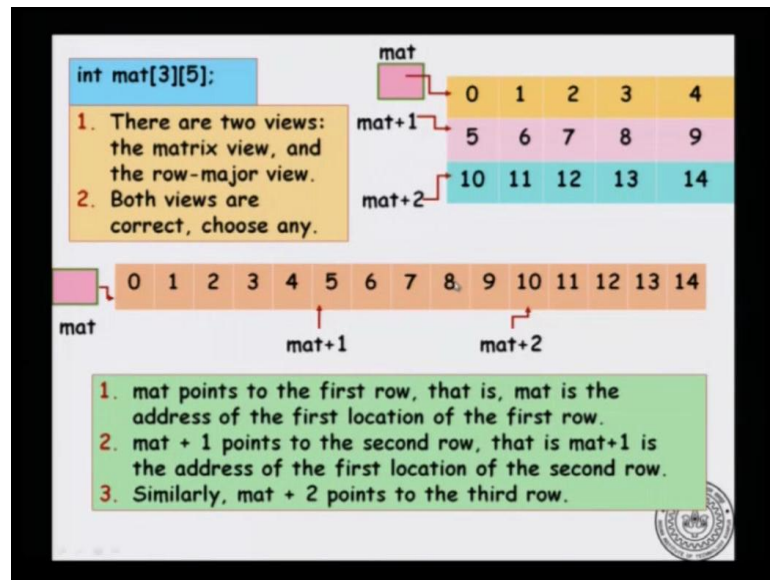


So, let us go back to, how do I address the  $i$   $j$  th element in a 2D array? Now, we can view it as a 3 by 5 matrix of integers, something like this. So, it may be an array 0 1 2 3 4, that is row 0 and 5 6 7 8 9, that is row 1, 10 11 12 13 14, that is row 2. So, this is the matrix view which is 3 rows, each with 5 columns, this is the standard view. But, internally C views this as a long linear array of size 15, in what is known as the row major form.

So, let us just look at what it is? Internally, C looks as loops at the array in the following form. It is basically C 0 through 14, lead out in a single row. So, this is the row major view, it is called row major, because first all elements of row 0 will be lead out, then all elements of row 1 will be lead out. And finally, all invents of the lost row will be lead out. But, it is lead out as a linear way.

Now, the natural question to ask is, in that case is a 2D array really at the heart of just a 1D array. So, the difference between a 2D array seen in the row major view point and an actual one dimensional array will come in the pointer arithmetic.

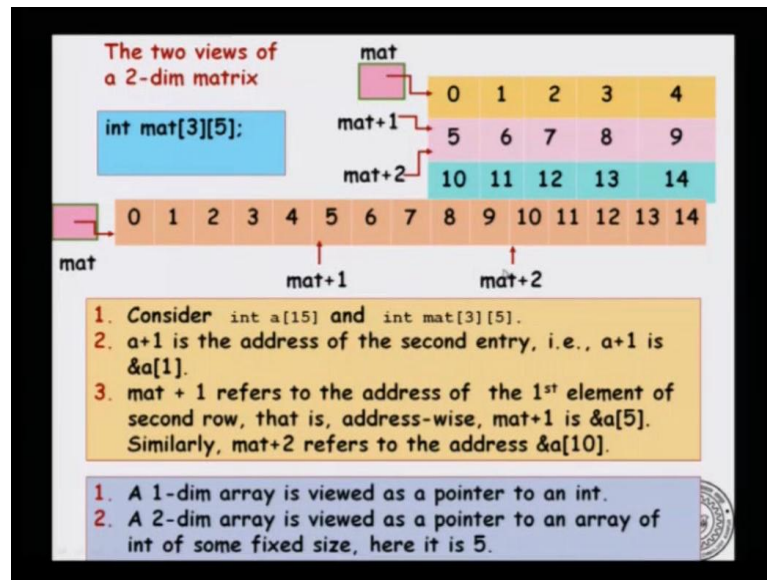
(Refer Slide Time: 04:52)



So, as I just mentioned there are two views, the matrix view and the row major view and both views are correct. So, if I have the matrix view, `mat` is a pointer to the first row. So, `mat plus 1` will be a pointer to the second row and `mat plus 2` will be a pointer to the third row. So, row number 3 or row indexed with 2, in the row major view point, here is the difference `mat` points to the first row and `mat plus 1` should point to the second row.

So, we cannot say that `mat` is pointing to the first element here and `mat plus 1` should point to therefore 1. Now, that is not what happens? It has to be consistent with the matrix view. So, the pointer arithmetic `mat plus 1` should point to the same element, regardless of whether you are looking at it using the matrix view point or whether you are looking at it using the row major view point. So, `mat plus 1` will still point to 5 and `mat plus 2` will still point to 10. So, these two view points are consistent.

(Refer Slide Time: 06:11)



Now, here is the difference with one dimensional arrays. So, we have just repeated the viewpoints here, the matrix view point and the row major view point. Now, had `mat` actually being a one dimensional array, `mat` would point to the first element in the array. Therefore, `mat + 1` which should point to the second element in the array. So, that is not what happens, it is actually the row major representation of a 2D array and `mat + 1` should skip exactly 5 elements, because that is the size of the column.

So, `mat + 1` should skip 5 elements and go to the element `mat + 5`. So, here is we just of why you need to know the number of columns? Because, in the row major view point I have to implement `mat + 1`. So, I have to say how many elements should I skip, in order to get to the first element of the second row? And that number is exactly the number of columns in the array. So, the number of columns in the array is 5. So, to get to `mat + 5` from `mat`, I would to skip 5 elements.

Similarly, to get to `mat + 10` from `mat + 5`, I would to skip exactly 5 elements. So, this is the reason why the number of columns is an important information. Because, that tells me in the row major representation, how many elements do I have to skip in order to get to the correct entry in this second row or third row? So, here is the pointer arithmetic for the row major representation. And notice that, this is considerably difference from the pointer arithmetic for a 1D array, in a 1D array, `array + 1` will go here, the first element of the array.

(Refer Slide Time: 08:12)

mat

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

mat+1 mat+2

A 2-dim array is viewed as a pointer to an array of int of some fixed size, here it is 5.

Can you guess what the type of mat would be?  
Which one below looks most right?

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

Recall that array indexing operator [] has higher precedence than \*, the dereferencing operator

int \* arr; (\*mat [5]);

float arr[5];

Now, can you try to guess what will be the type of mat? So, here are four candidates and let us go through them to see, what is the most likely candidate? And we will see this in a greater detail, int star mat, mat is a pointer to int. Now, we have seen that, that is approximately an array of integers and that is definitely incorrect. Because, this is supposed to be a 2D array, not an array of integers. Pointer to pointer to mat, we have seen that so far and that looks like a likely candidate.

So, what about the third and the fourth? The third and the fourth looks confusingly similar. What do they mean? So, here is a hint, the array indexing operator square brackets has higher precedence than star, the dereferencing operator. So, in this case the first says that, so what is this mean? The first declaration is actually int star mat 5 and the second declaration is int star mat 5. So, what does this say?

So, let us compare with the standard declaration like float array 5. This means, that array is arr is an array of size 5, each entry of type float. Similarly, this means the matrix mat is an array of size 5, each entry being a pointer to integer. So, it will be some matrix like this, it has five elements and each of them is a pointer. So, here is the view point for declaration 3. Now, what about declaration 4?

(Refer Slide Time: 10:26)

mat

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

mat+1

mat+2

A 2-dim array is viewed as a pointer to an array of int of some fixed size, here it is 5.

Can you guess what the type of mat would be?  
Which one below looks most right?

```
int *mat;  
int **mat;  
int *mat[5];  
int (*mat)[5]; ✓
```

Recall that array indexing operator [] has higher precedence than \*, the dereferencing operator

int \*arr (mat [5]);

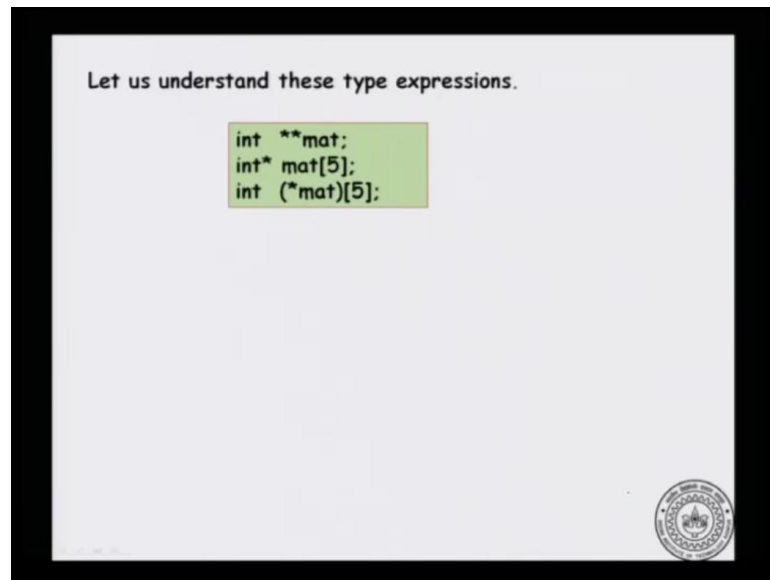
int arr [5];

So, there let us see this, so let us compare it with a standard declaration like, let us take a standard declaration like `int arr 5`. Again, this says that `arr` is an integer array of size 5. So, it contains 5 elements, each of type `int` correspondingly, what this means is that `star mat` is an integer array. So, here is an integer array containing 5 elements, these are integers. Now, this means that if we dereference `mat`. So, `mat` is a pointer to an array of size 5 and this is exactly the actual representation of a two dimensional array.

So, notice the difference between these two representations, the first says that `mat 5`, `mat` is an array of 5 entries and each entries is a pointer to an `int`, so it look like this. So, it is an array of 5 pointers to `int`. The lost declaration says that, `star mat` is an array of `int` of size 5. So, `mat` is a pointer to an array of integers of size 5. So, here is the difference and we will argue that the fourth definition is essentially what we want and we will see this in a greater detail.



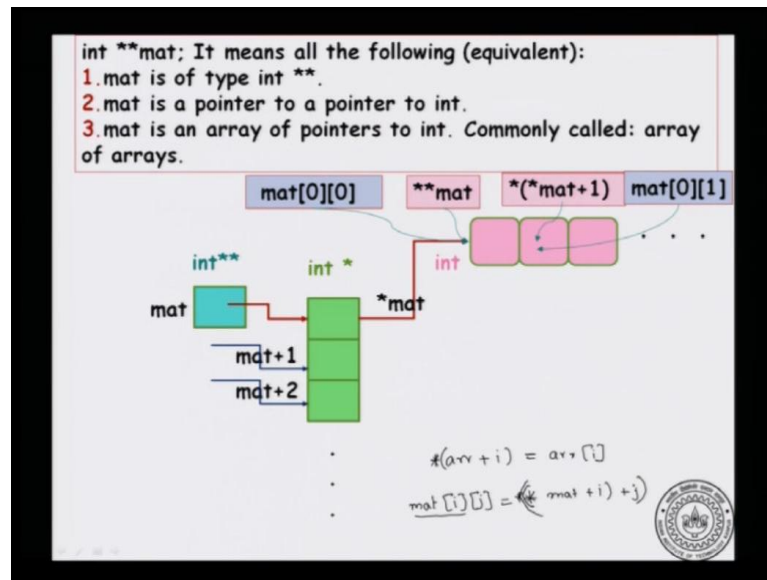
(Refer Slide Time: 12:24)



So, let us understand these type expressions in greater detail and we will see this in the further video also. We particularly pick on one representation here. So, we have argued that the 2D array is similar to the last declaration here, I eliminated the most obviously wrong declaration, which is in star mat, that is basically a one dimensional array. So, I have just eliminated that, we will examine all the others.

What I have just said is that a 2D array is similar to the last declaration. But, even the previous two declarations do make sense. And there may be situations, where you need to use such variables. Let us examine them in greater detail.

(Refer Slide Time: 13:05)



So, let us look at the first one which is `int star star mat` and it means all of the following equivalent ways. So, all of these are an equivalent ways of looking at the same thing. You could say that a matrix of type `int star star` or you could say that matrix is a pointer to a pointer to an `int`. Since, arrays are pointers approximately, you could also say that `mat` is an array of pointers to `int` and this is also commonly called array of arrays.

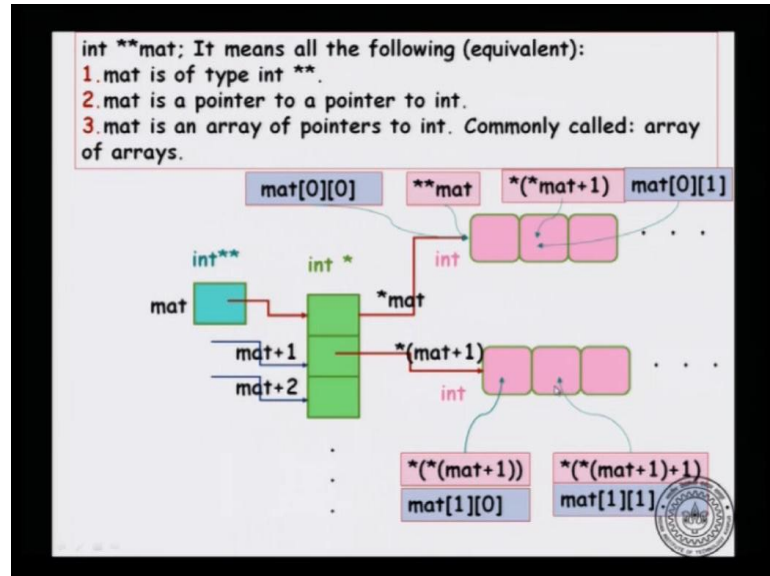
So, you have `mat int star star`. Now, this is a pointer to an array of integers. Now, every pointer to an `int` is essentially a pointer to an array, you can look at it like that. So, you can say that `mat` will point to an array of pointers and this array of pointers, each of them may point to a different array. So, you dereference `mat` once, you will get a pointer to an integer and again you dereference once more, you will get the actual array.

So, what happens when I do `mat plus 1`? It will go to the second entry in the array of integers. Now, that may be a different array all together. So, `mat 0 0` is similar to `star star mat`, this is just a way, address arithmetic works. And both of them are addresses, both of them are pointing to this location, both of them mean the content of this location. Similarly, `star of star mat plus 1` is `mat of 0 1`. So, in the case of one dimensional arrays we have just mentioned the equation that array of `i` is the same as `star of arr plus i`.

And what we are saying here essentially is that, `mat i j` is the same rule applied twice. So, I could say `mat i` is `star of mat plus i`. So, that will give me an array and then, I need the

jth element of that. So, you can again do star of star mat plus i plus j. So, these are two ways of looking at this array.

(Refer Slide Time: 15:50)



So, `mat + 1` will be the next element in the pointer to integers and it is the same as and when you dereference it, you will get another array. So, in order to get the first element of this array, I could say `mat[1][0]` or using the pointer notation, I have star of star `mat + 1`, these are the same and similarly for other elements of the array. So, one of the advantages of this kind of `int star star mat` is that, I have freedom in both dimensions.

You can see these as the rows of a matrix and these as the columns of a matrix. If you see that, then you can see that I have a lot of freedom here, first of all the number of rows is not limited. Because, it is just `int star star mat`, I could have any number of rows here. Now, another main advantage and the reason why this is somewhat popular is that, the length of row 0 need not be the same as the length of row 1, these are just pointers to integers.

So, the first pointer to integer may be pointing to a row of size 2, the second pointer may be pointing to a row size 3 and so on. So, the row lengths need not be the same. So, think of an array where row 0 is two elements long and row 1 has three elements in the row and so on. So, if you have extremely ragged arrays, then `int star star mat` is a nice representation to take.