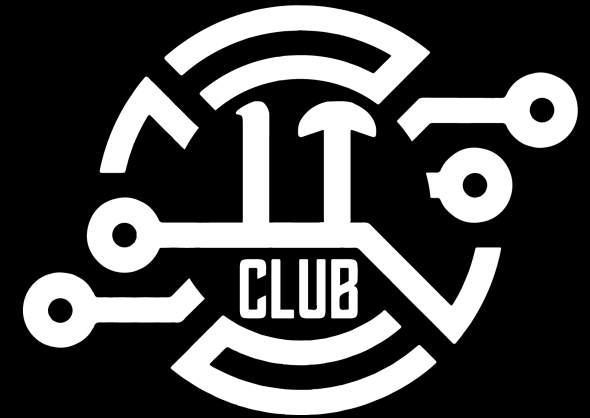


A WORKSHOP ON

THE C PROGRAMMING LANGUAGE



LECTURE 3

Inde-STRUCT-ible

Why Arrays?

LECTURE 3

Inde-STRUCT-ible

What is an Array?

An array is a collection of items of a **particular data type**, given a common name. Arrays are referred to as derived data types, as an array is derived from the basic data types.

```
int numbers[5]; //declares an array of integer type
```

```
float numbers[5]={2.1f,3.2f,7.9f,6.5f,4.21f}; //initializes an array with 5 float values
```

↓
Data type of
items

↓
Size

↓
Elements inside
curly braces



Accessing Items

An array item or element can be accessed by using subscript notations i.e. indexes. Arrays are indexed data types, thus an item can be accessed by its index. Note that index of an array **always starts at 0**.

```
int main() {  
    int a[5] = { 32, 34, 67, 89, 21 }; //can also use a[]={...}  
    int b = a[0];  
    int c = a[2];  
    return 0;  
}
```

Diagram illustrating array indexing for the array `a`:

	0	1	2	3	4
	↓	↓	↓	↓	↓
<code>a[0]</code>	32	34	67	89	21



A-a-a



Its first word!



Arrays start at 1



Array Items are Mutable

The items in an array can also be modified when need be. A value can be assigned to any index within the array. The previous value at the index is overwritten after the assignment.

```
int numbers[5]={21,32,79,65,421};  
numbers[0]=10; //array is now {10,32,79,65,421}  
numbers[4]=100; //array is now {10,32,79,65,100}
```



Looping Through an Array

Many tasks require us to check all the elements of an array, and perform certain tasks based on the required output. Manually using the index to get each item from the array, and then perform the task is not practical. Thus, we can use loops to loop through an array, and retrieve the elements without much hassle.

```
int main() {  
    int num[5] = { 1,2,3,4,5 }, i;  
    for (i = 0; i < 5; i++) {  
        num[i] = num[i] + 2; //adds 2 to every array element  
    }  
    return 0;  
}
```



Two Dimensional Arrays

Most of us already know what a two dimensional array is. We've been using it for a long time now in Maths.

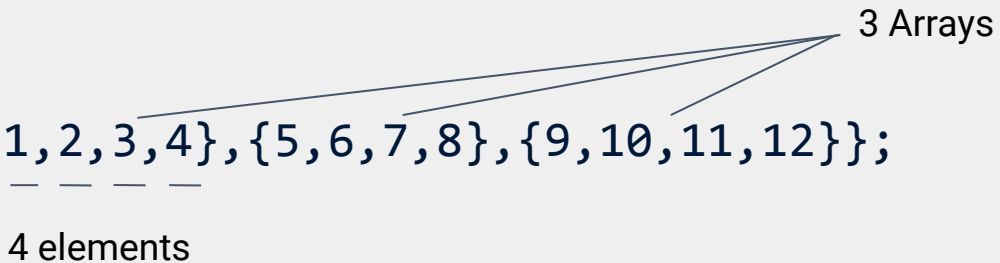
A 2-D array = Matrix

No, this is not a Maths class. No, I won't bore you with inverses and determinants. In C, a two dimensional array is an array that may contain one or more arrays as their elements, i.e. an array within an array.

```
int matrix_reloaded[3][4]={{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

4 elements

3 Arrays



Rearranging this two dimensional array,

```
int matrix_reloaded[3][4]={1, 2, 3, 4},  
                           {5, 6, 7, 8},  
                           {9,10,11,12}};
```

This looks like a matrix with 3 rows and 4 columns, with the arrays as rows and elements inside the arrays as columns.

Thus, a two dimensional array with a size of [3][4] represents a matrix of order 3x4.



Accessing 2D Array Elements

Accessing elements from a two dimensional array is quite similar to accessing an element from a one dimensional array. The only difference is that we have to provide two indexes to get a single item (similar to how we declared a 2D array). Both the indexes start at 0.

	Column 0	Column 1	Column 2	Column 3
Row 0	1	2	3	4
Row 1	5	6	7	8
Row 2	9	10	11	12



```
int main(){
    int matrix_reloaded[3][4]={1, 2, 3, 4},
                                {5, 6, 7, 8},
                                {9,10,11,12}};

    int a=matrix_reloaded[1][1]; // a=6
    int b=matrix_reloaded[2][0]; // b=9
    ..
    ..
    ..
    return 0;
}
```



Looping Through a 2D Array

```
int main() {  
    int i,j,a,matrix_reloaded[3][4] = { {1, 2, 3, 4},  
                                          {5, 6, 7, 8},  
                                          {9,10,11,12}};  
  
    for (i = 0; i < 3; i++) {  
        for (j = 0; j < 4; j++) {  
            matrix_reloaded[i][j]= matrix_reloaded[i][j]*2;  
        }  
    }  
    return 0;  
}
```



Others : why do you always
use i,j variabes in loops?

Programmers :



Why Structures?

LECTURE 3

Inde-STRUCT-ible

What is a Structure?

Structures in C provide a basis of creating user defined data types. Meaning, it is possible to construct a data type tailored to the user's needs, using the basic data types that are available. We can then create variables of this custom data type and apply everything that we've learnt to this data type.

```
struct name_of_struct{  
    datatype1;  
    datatype2;  
    ..  
    ..  
};
```

OR

```
typedef struct  
{  
    datatype1;  
    datatype 2;  
    ..  
    ..  
}name_of_struct;
```



Example

```
struct student{  
    int roll;  
    int marks[5];  
    float percent;  
};
```

OR

```
typedef struct  
{  
    int roll;  
    int marks[5];  
    float percent;  
}Student;
```

Here, a structure named 'student' is created. Basically we have created a new data type, using the basic data types available to us. We can now declare objects(variables) of this data type. The potential to customize and create a data type of your own is the reason why it is called a **user defined data type**.

Structs can be declared either inside or outside the main() function.



Declaring Objects

```
struct Student{  
    int roll;  
    int marks[5];  
    float percent;  
};
```

OR

```
int main(){  
    struct Student a,b,c;  
    ..  
    ..  
    return 0;  
}
```

```
struct Student{  
    int roll;  
    int marks[5];  
    float percent;  
}a,b,c;
```

OR

```
int main(){  
    ..  
    .. // can use a,b,c in main  
    ..  
    return 0;  
}
```

```
typedef struct  
{  
    int roll;  
    int marks[5];  
    float percent;  
}Student;
```

```
int main(){  
    Student a,b,c;  
    ..  
    ..  
    return 0;  
}
```



Initializing a Structure Object

You can give values to the structure objects by including them in curly braces, and writing the data serially.

Here, **2** corresponds to **roll**
{33,44,55,66,77} corresponds to **marks**
83.0f corresponds to **percent**

```
struct Student {  
    int roll;  
    int marks[5];  
    float percent;  
};  
  
int main() {  
    struct Student a={  
        2,{33,44,55,66,77},83.0f  
    };  
  
    return 0;  
}
```



Accessing Members of a Structure

Accessing a member (the basic data of the structure) through the structure variable is achieved by the **. (dot) operator**. All the basic variables declared inside of the structure can be accessed using this operator.

```
struct Student {  
    int roll;  
    int marks[5];  
    float percent;  
};  
  
int main() {  
    struct Student a={  
        2,{33,44,55,66,77}  
    };  
    int b = a.roll;  
    int c = a.marks[0];  
    float d = a.percent;  
    return 0;  
}
```



Array of Structures

Just like we created arrays using basic data types (int, float, etc.), we can also create an array of a structure. Doing so helps us to store a large amount of information in a relatively small amount of variables.

a[0]	a[1]	a[2]	a[3]	a[4]
int roll int marks[5] float percent	int roll int marks[5] float percent	int roll int marks[5] float percent	int roll int marks[5] float percent	int roll int marks[5] float percent



```
struct Student {  
    int roll;  
    int marks[5];  
    float percent;  
};  
  
int main() {  
    struct Student a[5] = { {1,{95,95,95,97,87}} };  
    int i, total = 0;  
    for (i = 0; i < 5; i++) {  
        total += a[0].marks[i];  
    }  
    a[0].percent = total / 5.0f;  
    return 0;  
}
```



Exercises

- Initialize a 2-Dimensional Matrix, and find the element with the maximum value.
- Declare a structure named '**complex_no**', which represents a complex number with a real and imaginary part. Create two variables of '**complex_no**', add the complex numbers, and store the result of addition to two variables namely '**real_part**' and '**img_part**'.

LECTURE 3

Inde-STRUCT-ible