Lec 02 Module 01 Recap of C (Lecture 02)

C++ Object oriented (Indian Institute of Technology Kharagpur)
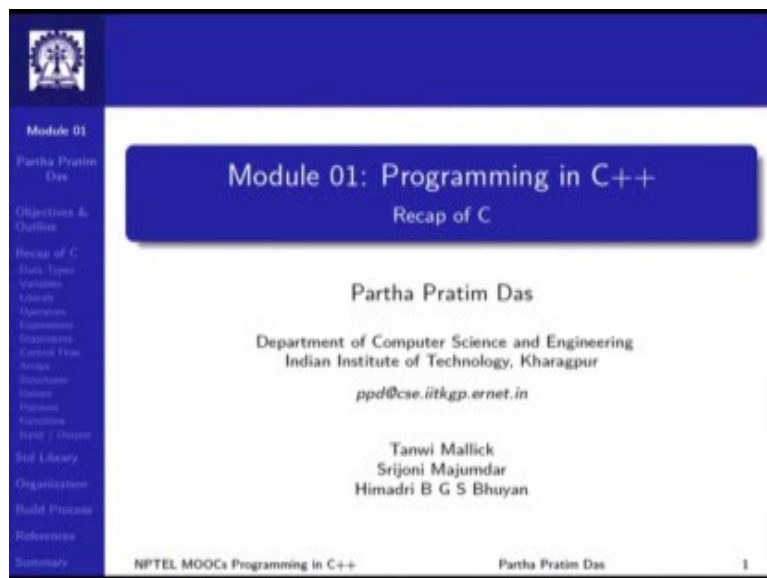
Lecture - 02
Recap of C (Part II)

We have been discussing about C programming in Module 1. This is the part-2, we will now talk about.

(Refer Slide Time: 00:25)

In the earlier part you have seen the basic concepts elementary concepts of C including data types, variables, operators, expressions and statements particularly the control statements. Now, we will move on and we will talk about various derived types in C.

(Refer Slide Time: 00:59)

As you know the most common derived type in C are arrays. Array is a collection of data items. So, a variable is a single data item, it is a single value; where is array is a collection of one or more data items with the restriction that all data items have to be of the same type. So, if I say there is an array of int, then everything in that all value in that array has to be int. If I say that array is of type char, then every element has to be of type character.

In terms of arrays, the first thing we need to do is to declare an array like we need to declare any variable, so we declare the array in this. So, naturally, we need to specify the name; we need to say what is the type of the element of an array; and also we need to specify what is the maximum number of elements that an array can support.

Now, this maximum number of elements can be specified in multiple different ways. And as we will see that this is one aspect in which arrays in C++ will somewhat differ from arrays in C, but a typical array in C will be defined as double balance within corner brackets 10, which will say that balance is an array of double type elements and maximum 10 such elements could be there. Otherwise, we can define a size by a manifest constant and use that to define declare the array.

Then array can also be initialized, we are moving to the second part. So, arrays can also be initialized that is if I just declare an array then the elements do not have any specified value, but I can declare as well as initialize the array. So, if I say 'int primes[5]' initialized with this list of values which are separated by comma and is content within a pair of curly braces then the first value in that list goes to the first element of the array which is the 0th index element as you know.

The second on the list goes to the second element, which is at the index one and so on. Interestingly in C, arrays are allowed to be initialized even without a specifically express size. So, I can just write primes without actually giving a size and just give the initialization list what C compiler does; it finds out how many elements you have initialized it with, and assumes that is the size of the array or those many elements will be there.

So, if primes are just initialized with the list of 2, 3, 5, 7 and 11 there are 5 elements, so it will become an array of five elements. And I have also shown in such cases, since you do not know what is the size, what is the most reliable way to compute the size that is you find what is the total size of the array divided by the size of every element you will certainly get the number of elements that the array contains. It is also possible that you have a bigger array and initialize it with less number of elements; the initialization will always happen from the beginning.
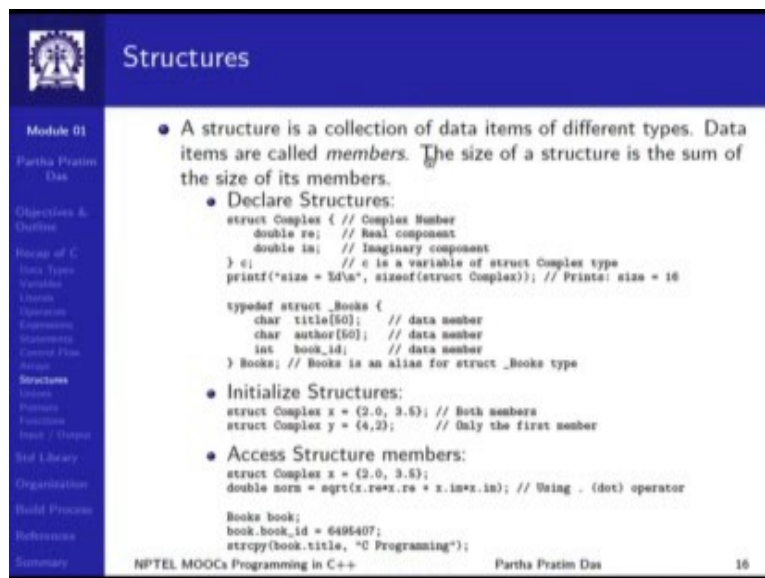
And the remaining elements in such cases will be initialized to 0, but you cannot in it have an initialization list which is bigger than the size of the array that will be an error. An array whether it is initialized or not initialized can be accessed by each and every element. So, next for accessing array elements, we can see that we use the index and we can read or access that element, which exist at that array location. Similarly, I could not also write it on the left hand side of an expression and make an assignment to an array element.

C assumes that every array as it is defined is like a single element also. So, I can define arrays of arrays and of arrays and of arrays and so on; and in this way, I can extend into multiple dimensions. So, these are called multidimensional arrays. So, the simplest of the multidimensional array would be a two-dimensional array, which we commonly in mathematics, we commonly call them as matrix. So, it is given with two dimensions; one is

the number of rows and the number of columns. So, if we define int mat [3][4] that mean, that there are 3 rows and 4 columns in this whole mat array that we have.

So, it makes it two-dimensional; naturally if it is two dimensional, it needs two indices the row index and the column index to be accessed. You can extend it to 3, 4 any higher dimension as it is required, but it is less common that you will have 3 or higher dimensional arrays in regular use.

(Refer Slide Time: 06:35)

Next to arrays are structures. So, like array is a collection and we will slowly start using the term container a term which is not so common in C, but we will see in C++ and particularly the standard library that comes with C++, the terms commonly used is called container because container is something which can contain other elements. Array is a container; it is a collection of data items with the specific behavior that all items are of the same type as we have seen.

Now, structure in contrast is also a container; it is a collection of data items, but here the data items could be of different types; it is not necessary that they have to be of different types, but they can be of different types. So, if I look into a structure, these data items are often called

members; and we will more and more use this term member or data member to mean components or data items of a structure, we show a simple example of forming a complex number.

As we know complex numbers have two components - the real part and the imaginary part. So, each part can be a double. So, we define that with struct complex that tells us that there are two components; in this case, both of them are of the same type and then C is declared to be a variable of this structure type. We can also, in the next one; we show that you truly have one structure where components are of different type. So, the first two components in the books structure are title and author; they themselves are arrays of character which is basically what it means that they will become C strings of title name and the author name; and the third component is actually an integer, which is keeping the book id.

Structures could be defined directly as by their name, and using the keyword struct we can use that or the struct and the structure name can together be given and alias by the use of type def keyword. Type def is a short form of type definition. And as we go into C++, we will talk more about this that why type def actually is not a type definition, it is kind of a misnomer, but historically it has this keyword has been used, and it continue to be used in C, but it is basically type alias. It is just another name given to the struct complex of the struct books, as we have. It just becomes it easier.

If you use, have a type def then you can directly use it. So, if we come to the access of the structures then we can see that since complex was just defined as struct complex to use that we need to write struct complex and then the variable name followed by the initialization of course. But books here was defined by type def, so books does not need to be written with struct books I can just write books, it has given a total name for the struct book. So, it becomes easier to use it in that way. Now, any structure variable can be initialized like a simple variable; and the notation is very similar to the way we initialized arrays.

In contrast, in arrays, the list formed within curly braces meant different components or different indexed elements of the array here; in case of structure, it means the different components or the data members of the array; and the way you list these initialization values, at the way the data members are listed from the top to bottom order. So, if we look into the

specific cases given in this code, then you have we are showing struct complex x initialized with 2.0, 3.5, which means 2.0 will be the initial value of the first component which is 're'. So, you just read them from top to bottom and read this list from left to right and correspond to them. So, 3.5 will be the initial value of 'im'. So, x will get become a complex number 2.0 + j 3.5, if we think of it that way.
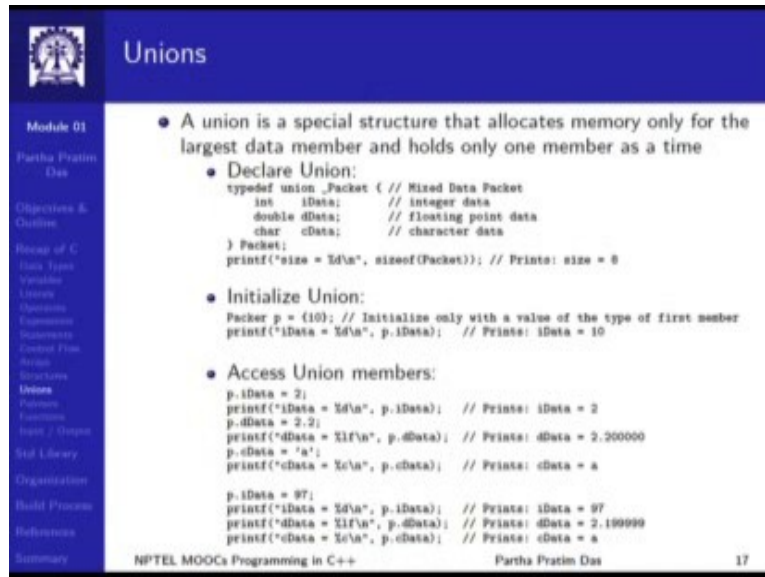
It is possible that we initialize only one or some of the members, but you can do that only from the initial part starting from the top. You cannot skip 're' and just initialize 'im', but you can initialize 're' as you are doing here with 4.2 and skip 'im', which is not mentioned here. Then every component of a structure can be accessed by the dot notation, the dot operator we are showing here. So, if x is a complex variable, a structure variable, and then the 're' component of this variable x is written as x.re. Similarly here book is a structure of the above type books type, so it has an 'id' component which is book.id. So, we use the dot notation to access the structure component.

So, we can see among the two most commonly used, two most powerful containers in C have two different ways of access; array since all elements are of uniform type. It is accessed by position, it is called the positional access, because you want to find out in the array starting from the beginning the zeroth, first, second, so you just go by number, so array is also called a indexed container, because you can access the elements by number.

In contrast, in structure, the elements are of different components are of different types. So, there is not much significance to what is a first component in the list or what is a third component in the list and so on. So, you access the elements by name. Now this is also a style, which we will see in different languages as to whether you can access something by name or you can access something by position, and similar issues of access will come up when we talk about function arguments also which in C as you know is also by position.

And I would just like to I would like that these are few common aspects of programming languages that you need to be careful about as to when you have a list of elements what is the access mechanism between whether it is positional or it is by name. So, in structure, we see one kind; in array, we see the other.

(Refer Slide Time: 13:54)

Now moving on a C supports another kind of collection container, which is pretty much like structures, they are called union. Now, the only difference is instead of using the struct keyword you use the union keyword. The way you specify or way you access the members are same between structure and union. But what is different is the way the memory is allocated. If you have a structure with two, three different members then when a variable of that structure type is created all members are allocated area. So, if you go back to this structure, then you will see that here after the complex structure has been defined it has two components. So, both these components are actually allocated.

So, if we assume that every double texts bytes then the size of the whole complex structure is 8 bytes plus 8 bytes that is 16 bytes, because both exist. In contrast, in union, it is the allocation is done in memory for only the largest component. So, you make sure that your basic assumption is as name union suggests that this is a union of all these fields, so at any point of time only of the components will exist only one of the components has to be assumed. So, naturally if you are representing only one of them you need enough space for the largest one that is only the logical thing. So, if we look into this union packet then we will see that in packet, 3 components are of int double and char type.

So, if we take a reasonable assumption about the size in a say 32-bit machine then int will be 4 bytes, double will be 8 bytes and char possibly will be 2 bytes. So, the size of packet will be 8 bytes which is the size of the largest component double. So, the interesting aspect is this on one side allows you to minimize the size of the whole record, the size of the whole container, if you know that you do not need all the components at the same time.

So the flip side is when you initialize you were initializing only the value of the first component because certainly here since you can have only one component, we have space for only one component the initialization cannot have a list of components list of constants to initialize different data members. And we have seen that in initialization can be partial provided I have initialized the first one or the first two or the first three here since there is only one data member there is only one value that you can initialize and that has to be the type of the first value.

Otherwise, you will have to you will not need to initialize rather you will simply take the component and assign that. So, in terms of access, the same dot notation the dot operator will be useful for accessing the different components, but the point is you will have to remember that only one value is maintained out of though we have three components there is only one value.

So, depending on what is the component that you have assigned last when you access the value of that component only will be valid. If you access a different component then what was assigned last then you will get unexpected result. So, in this access code if you look at little carefully, then you will see in the first part of the code we saw that iData the integer we assigned to and then we access iData, so 2 is printed. In double data, we do 2.5 access double data 2.2, so double data we access the 2.2 is printed. In the char data, we assign constant character 'a', we access that 'a' is printed. So, we whatever you access last are what are printed.

Now, in the next, we show an interesting thing we in the iData there is a integer part we assigned 97, and then we accessed the integer data; obviously, 97 is printed the last. But if without assigning anything we access it as dData then you get some 2.1999 which is not something very meaningful. So, this is happening because the dData, double data has much

larger size 8 bytes; whereas you had assigned iData which means that you have only assigned 4 bytes of that; remaining 4 bytes have some old garbage value. So, what you get as dData is completely wrong.
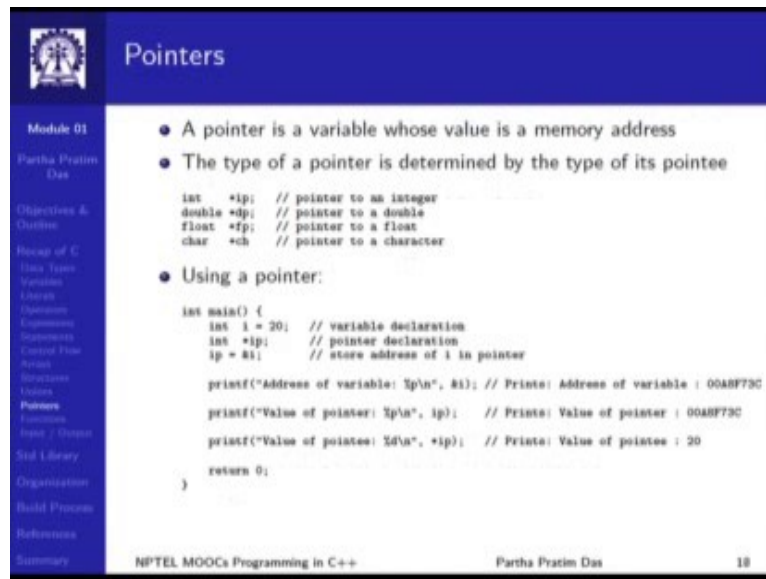
The final interesting thing is if you access it as C data then you are actually accessing 97 as C data, and you see you are getting 'a' why, 97 is the ASCII code of 'a'. Now the interesting thing is that C data possibly is 1 byte here, which means that it can contain up to 255 the value incidentally that you have given to the 4 bytes of int happens to be less than 255. So, we can easily understand that in this int the higher three bytes are all zeros. So, when I access it as a character I just get the 97 that I have stored there. So, it feels as if I have got the corrected value which is the code of a. The point that I am trying to highlight is it is one value that resides, because there is only one memory location; you will have to be very careful about this when you are using union.

The reason union was given in C is for the fact that if you have to make an aggregate, if you have to make a container which can deal with data of variety of types, typically just to give you the kind of reference where union is used is if you think of a network connection, and you have a network port, where variety of data is coming different kinds of data packets are coming, and you do not know what kind of data packet will come out of a possible say 10 different kinds of data packets.

But you know that at any point of time only one kind of data packet will come, only one data packet will come then how do you define a programming structure to store this data packet. Now, you do not want to make a structure which has the possibility of storing all these 10 different kinds of data packets, because that will unnecessarily take a lot of space. But at the same time, utilizing the fact that only one type of packet will come at 1.0 of time you could use this union structure as a collection of also that is how it has been used in C.

And we will see in terms going to C++ there will be a lot of ramifications of this concept of union C++ does provide us with a very strong feature based on object orientation something known as inheritance and specialization. And we will see how similar effects can be created in C++ without using union, so that is just for your comments, we will come to with the details subsequently when we go to that level of C++.

(Refer Slide Time: 21:55)

Now next the most interesting and possibly one of the most powerful feature of C programming that comes up. C programming, if you are familiar with little bit of the history of the language, and I seek this to be an appropriate point to introduce that to you, it is before C was done in an university by a group of computer scientists, professors typically, and a group of professionals, we were trying to write an operating system which later on became popular as Unix, you all use Linux which is a much later generation of that. So, while they were trying to write Unix, trying to write code for Unix, they needed a programming language, because otherwise how do you write code.

Now prior to Unix, there was no high level programming language of the type that you see in C or C++ or java or python available, where you could write an operating system, because when you want to write an operating system, you need to not only deal with values, but also you need to deal with memory. Because the programs will finally, get stored in memory that where data will reside in memory. So, when we write a program, we are just dealing with variables, we are just interested about values. We can write a complete program like in python or in java without at all thinking of where these values are stored in memory, but you cannot write an operating system assuming that.

You need to that programming language to be aware of the memory or to be aware of the address where values get stored. So that was one of the various reasons for which the team of Kerning, Ritchie, Morris buck and all others, we were in the Unix team needed to do a programming language, and they quickly designed C which then later on became a default language for all of us, rest of it as they say is history but this is the genesis for which C for the first time introduced the strong concept of managing addresses as data and that is what we say is a pointer.

I just wanted to give you the reason why you needed this, but when you have this feature of dealing with address as data that gives C a very strong background; in which C can create variety of data structures as you all some of you all have already done. Like you cannot think of doing a list without having pointers; it is possible, it is possible you can have create a link list by using two arrays. An array to keep the index the address of where you will find the next element; and another array to actually have the values, but that is not something, which is efficient which is scalable which what people would do; so, you will always use pointers.

So, pointers are the next derived types as you all know. So, it is a variable whose value is a memory address. And the type of a pointer is decided by not by the pointer itself, all pointers are addresses. So, address which has only one type that is of the pointed type, but their type is decided based on what kind of value they are actually pointing to. So, if I have an int *ip then it is pointing to an integer type of value.

So, we will say it is int * type. Now, to use pointer, you will be familiar with this then I can have a variable i which is initialize to 20, I have a pointer ip which I would use to point to i, which is int *ip and there is a special operator ampersand as you know which I can use with any variable to get the address of the memory location, where this variable will be stored. So, I can take that address and store that address in the pointer variable ip.

And once I have done that then if I try to print the address of variable i or i print the address or the value of ip, which is actually the address that I have stored here then they will certainly be identical that is what we are showing here. Now given that address, I can actually find out what value exists at that pointed location by using the star or content of operator. So, here along with the pointer, we will always use the address of as well as content of operators.

(Refer Slide Time: 26:49)

Now pointers can be used in multiple different ways; they had very powerful in terms of creating various idioms in the programming language. The first and the most common is the duality between the pointer and an array; and most easily seen in terms of a one dimensional array. So, array is a series of locations; and pointer is address of the starting location of the array. So, I can take the array and assign it to a pointer variable. And then if I just do *p, it gives me the content which will be the content at the starting location of the array which will happen to be a 0.

I can increment a pointer and that is a very interesting concept, if I increment a pointer then the amount by which it is incremented is not 1, the amount by which it is incremented is the size of the type of element it is pointing to type of value it is pointing to. So, if it is pointing to an int and the sizeof(int) in the system is 4, then the pointer value actually will increment by 4, so that in terms of the array now you are pointing to the second location. So, you can see that if I do * + + p, it will first increment.

So, it is now pointing to element 1 and then it takes the element 1, which is basically 2. Similarly, I can have I can take p + 1 as an expression, p + 1; similarly is the current location of the pointer plus 1 element size what that can be. Pointers can be used with structures, if we

use that then we can access the elements by *p, and then *p is a structure that it is pointing to dot re is a component of that structure.

This can be shortened by a de referencing operator as this given in C, you know all this. So, we will skip that then pointers can also used in terms of dynamic allocation. So, I can dynamically allocate using malloc, and I get a pointer which does not have a specified type we could say that it is a void star. And we can use a casting which is forcibly to be done by the programmer to cast that to integer type value. This is an interesting code given here, I will not go through the details; try to understand how this code works how to manipulate with pointers. And if you have questions, you can ask us on the blog. And we can use pointers for dynamically allocating arrays as well.

With this, in this part of the recap, we have primarily talked of the different derived types. First, we have talked about the containers, arrays, structures and unions; the three main types of containers that C provide; and we have talked of managing the different variables and addresses through pointers. In the next, we will talk about the functions.