Structure

Title Slide

Array is the compound data type we have seen. One of the limiting factors in case of arrays is it can hold only homogeneous data type. You cannot conceive of an array holding integers and floats together. Having seen arrays and pointers, the homogeneity of arrays is very important.

Nevertheless, in real life we need to hold data of different types together that may represent single entity. Imagine that you are writing a C program for student information processing. A student entity has different characteristics viz. Name, department, gender, roll no. Etc. None of the data types we have seen so far can hold all these information together as a single unit. It is a very bad idea to store all these information in separate variables. Then those separate variables need to be arrays to store information related to more than one student. Since arrays cannot grow or shrink, the dynamic nature of student list will pose serious issues. We are definitely in need of a compound data type that can hold heterogenous data types together. **Structure** is a compound data type in C designed precisely for this purpose.

As the picture shows `number` is a structure that can hold both an integer and a float. As numbers can be of both types. This is an abstract example. The syntax of the structure as you can see in this picture is trivial.

Slide 2: Syntax

This slide explicity makes the syntax clear. `struct` is the keyword used to declare a structure. This is followed by the name and a pair of parantheses that hold the heterogeneous collection of data. As with any C statement, ; (semicolon) delimits the declaration.

Slide 3: Example

This slide shows an example declaration. The structure `person` contains a 1D string, character, integer and 2D strings. The names of the variables declared inside `person` makes the context clear.

The name of the structure `person` in fact is an user-defined structure. Using this user-defined structure you can create any number of `person` instances!!! The next slide shows this.

Slide 4: Variable Declaration

If you compare slides 3 and 4, you can see that there are two variable names `person1` and `person2` appearing before the delimiting ; (semicolon).

The variable declaration, shown in this slide, declares a variable `person1` of data type `person` (which essentially is a structure) and declares an array `person2`. The array is an array of 20 structures of type `person`. `person2` can hold aadhar information of 20

persons (it is an array of structures!!!). Ofcourse `person1` can hold the aadhar information of a single person.

Slide 5: Variable Declaration

This slide shows another possibility of structure variable declaration. Assuming that the structure `person` has been defined above the main (you can do that), the variables `person1` and `person2` has been declared using `struct person` as the keyword.

Essentially it means that the structure variable names need not always be part of the structure definition, it could be separate.

Slide 6: Accessing Members

Assuming that `person1` is a variable of type `struct person`, we can access the variables inside the structure (which we will call members) using a dot notation. As this slide shows, the name of structure variable precedes the dot notation and the member variable succeeds the dot notation like `<struct_variable_name>.<member_name>`

The examples `studentInfo` and `addDistance` are two example C programs that will be attached along with this lecture slide and companion for your reference.

Slides 7 and 8: Custom Datatype and Variable Declaration

Slide 7 shows another interesting way to declare structure variable as a user-defined datatype. The keyword `typedef` creates a new (user-defined) datatype called `personnel`. Note that because of the keyword `typdef`, `personnel` is not a variable of type `struct person`, but a new datatype itself.

This is clear from the variable declaration shown in slide 8. Though Slide 5 and 8 achieves the same thing, one convenient thing with Slide 8 is the custom data type `personnel` which might convey the context clearly.

Slide 9: Nested Structures

This slide shows the definition of `employee` structure and declares a variable emp which essentially is an 1000 size array of structures. Interestingly, the example also shows that structures can be nested. The context and necessity of the `struct person` in the `struct employee` is apparent and self-explanatory. Note that the information designation and experience of an person is only relevant in terms of the employment of the particular person. In fact the `struct person` can be re-used in other appropriate contexts too.

An example dot notation to access member (i.e. `age`) of the nested structure i.e. `struct person` is shown in the bottom of the slide.

Slide 10: Structure Pointers

You can have pointers to structure too. The declaration of a structure pointer is straightforward as the slide shows. Rather than discussing the importance and necessity of structure pointers, this slide would like to leave it with the point that pointers can be declared for the structures too. In fact, once having understood the necessity of pointers in the context of arrays, it is straightforward to appreciate the importance in case of array of structures too.

Slide 11 and Slide 12: Union and Union Declaration

Union is another compound data type like structure. The parallel between the two is shown in this slide. The two compound data types are very similar such that just replacing the keyword `struct` with `union` is suffice to declare a union variable.

What then is the difference between the two?

Slide 13: Union vs Structure

Assume that a structure has as its members `name` (of size 32 bytes), `salary` (of size 4 bytes), `worker_no` (of size 4 bytes). Then a total of 40 bytes is allocated for the structure. However, interestingly, in case of union the size of the largest member (i.e `name`) is allocated i.e. only 32 bytes are allocated and the two other members `salary` and `worker_no` share the same space. This means only the last stored variable can be accessed in case of union as it overwrites other variables because the space is shared between variables.

Why do we need such a restrictive type then? - Unions are particularly useful in Embedded programming or in situations where direct access to the hardware/memory is needed. Unions allow data members which are mutually exclusive to share the same memory. This is quite important when memory is more scarce, such as in embedded systems. (credits: https://stackoverflow.com/questions/252552/why-do-we-need-c-unions)

Slide 14

This slide shows a practical example where structure finds application. For example this slide shows a structure to abstract the network packet header. The member names convey the necessary information needed to realize a packet. Even without the knowledge of networks, you can intuitively guess that the moment you ask for google.com the network packet has a source address (your local compute), destination address (google server), the type of message etc.

In fact you can also see a union nested inside the structure. Though we will not get into the details feel free to google for nestedness between structure and unions.

Slides 15-19

These slides show an example structure, structure declaration-cum-initialization, assigning a structure variable to another. The latter could come handy in dealing with situations shown in slide 18.