

# 18ES611

# Embedded System Programming

*Sarath tv*

# Structures

C **arrays** allow you to define type of variables that can **hold several data items of the same kind** but **structure** is another user defined data type available in C programming, which allows you to **combine data items of different kinds**.

A structure can be considered as a **template used for defining a collection of variables under a single name**. Structures help programmers to **group elements of different data types into a single logical unit**.

A single **structure** might **contain integer** elements, **floating**– point elements and **character** elements. **Pointers, arrays** and other **structures** can also be included as elements within a structure. The individual structure elements are referred to as **members**.

Since members are NOT the same type/size, *they are not as easy to access as array elements that are the same size.*

Structure variable names are NOT replaced with a pointer in an expression (like arrays)

# Defining a Structure

```
struct [structure tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```

## struct statement

The struct statement defines a **new data type**, with more than one member for your program.

The **structure tag** is optional and **each member definition is a normal variable definition**, such as `int i;` or `float f;` or any other valid variable definition.

At the **end of the structure's definition, before the final semicolon**, you can specify one or more structure variables but it is optional.

```
struct Books
{
    char    title[50];
    char    author[50];
    char    subject[100];
    int     book_id;
} book;
```

```
struct structure_name  
{  
data_type member1;  
data_type member2; ..  
data_type member;  
};
```

```
struct person  
{  
char name[50];  
int citNo;  
float salary;  
};
```

```
struct person person1, person2,  
person3[20];
```

This declaration above creates the derived data type struct **person**.

Another way of creating a structure variable is:

```
struct person  
{  
    char name[50];  
    int citNo;  
    float salary;  
} person1, person2, person3[20];
```

```
struct S {  
    int a;  
    float b;  
} x;
```

Declares x to be a structure having two members, a and b. In addition, the structure tag S is created for use in future declarations.

```
struct {  
    int a;  
    float b;  
} z;
```

Omitting the tag field; cannot create any more variables with the same type as z

```
struct S {  
    int a;  
    float b;  
};
```

Omitting the variable list defines the tag S for use in later declarations

# Keyword typedef while using structure

Writing **struct structure\_name variable\_name**; to declare a structure variable isn't intuitive as to what it signifies, and takes some considerable amount of development.

Developers generally use **typedef** to name the structure as a whole.

typedef keyword is used in creating a **type comp** (which is of type as struct complex).

Then, **two structure variables comp1 and comp2** are created by this comp type.



```
typedef struct
complex {
int imag;
float real;
} comp;

int main() {
comp comp1,
comp2;
}

ex=STR_2
```

# Accessing members of a structure

There are two types of operators used for accessing members of a structure.

- Member operator(.)

- Structure pointer operator(->)

Any member of a structure can be accessed as:

`structure_variable_name.member_name`



# Passing structures to a function

## Passing structure by value

A structure variable can be passed to the function as an argument as a normal variable.

If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

Example :STR\_arg

# Passing structure by reference

The **memory address of a structure variable** is passed to function while passing it by reference.

If structure is passed by reference, **changes made to the structure variable inside function definition reflects in the originally passed structure variable**

In this program, structure variables dist1 and dist2 are passed by value to the add function (because value of dist1 and dist2 does not need to be displayed in main function).

But, dist3 is passed by reference ,i.e, address of dist3 (&dist3) is passed as an argument. Due to this, the structure pointer variable d3 inside the add function points to the address of dist3 from the calling main function. So, any change made to the d3 variable is seen in dist3 variable in main function.

# C Programming Structure and Pointer

Structures can be created and accessed using pointers. A pointer variable of a structure can be created as below:

```
struct name {  
    member1;  
    member2;  
};  
int main()  
{  
    struct name *ptr;  
}
```

Here, the pointer variable of type struct name is created.

# Accessing structure's member through pointer

A structure's member can be accessed through pointer in two ways:

- Referencing pointer to another address to access memory

- Using dynamic memory allocation

- 1. Referencing pointer to another address to access the memory**

STR\_access\_reference

Using -> operator to access structure pointer member.

Structure pointer member can also be accessed using -> operator.

`(*personPtr).age` is same as `personPtr->age`

`(*personPtr).weight` is same as `personPtr->weight`

2. Accessing structure member through pointer using dynamic memory allocation

```
ptr = (cast-type*) malloc(byte-size)
```



**THANK YOU!!!!!!**