



C Programming

Lecture 7

Data Modifiers

*This material is developed by IMTSchool for educational use only  
All copyrights are reserved*

# Summary for Data types in C

## Data types in C

### Primitive

char  
int  
float  
double  
void

Function return void

Function takes void

Pointer to void

### Derived

array  
pointer  
function

### User Defined

struct  
union  
enum

## struct data type

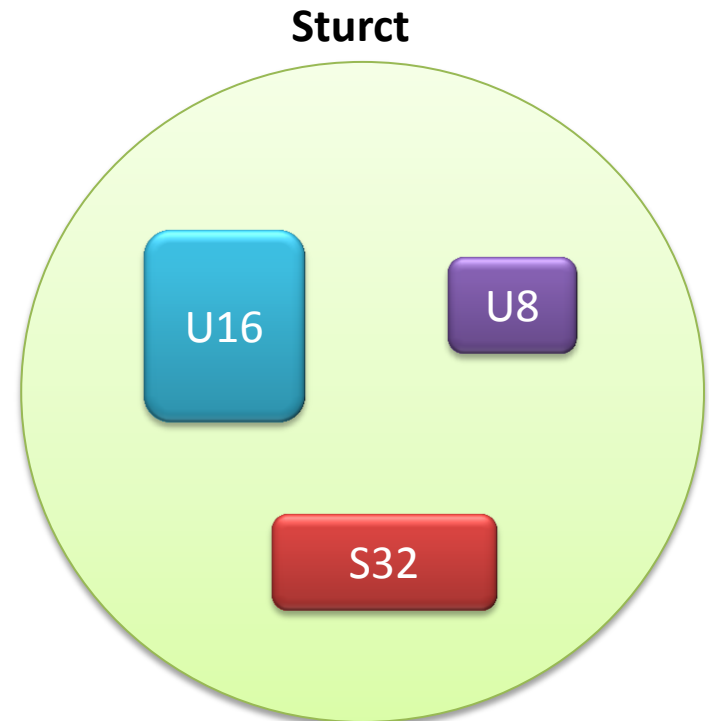
This data type contains many types under the same umbrella.

### Syntax

```
struct struct_name  
{  
    member_1_Type member_1_Name ;  
    member_2_Type member_2_Name ;  
    :  
    :  
};
```

### Note

**struct name is optional**



## Structure Example

This example creates a structure named *Employee*. This structure has two members one called *ID* and the other called *Salary*.

```
struct Employee
{
    u8    ID;
    u16   Salary;
}
```

### Note

**At this step, we didn't create a variable yet, we just declared a new type to the compiler. No memory consumed till now ... !**

## Creating an object from structure

### Syntax

**struct** struct\_name object\_name;  *Non initialized variable, it would have a garbage values*

Or

**struct** struct\_name object\_name = { member\_1\_Value , Member\_2\_Value, .... }

### Example

```
struct Employee Ahmed;
```

```
struct Employee Ali = { 11 , 3000 };
```

ID

Salary

## Accessing Structure elements

We use the *dot operator* ( `.` ) to access any member in the structure.

### Syntax

*object\_name* . *member\_name*

### Example

```
Ahmed.Salary = 5000 ;  
  
printf ("%d", Ahmed.ID) ;
```

**Note**, you can access all members of the structure at the same time only at the definition, otherwise you can only access elements one by one.

## Note

You can create some objects when declaring a structure at the same statement.

Note that the scope of these objects are the same as the scope of the structure itself.

In this example, Ahmed, Ali and Mohamed are objects from the structure Employee.

```
struct Employee
{
    u8    ID;
    u16   Salary;
}Ahmed, Ali, Mohamed;
```

Write a c code that defines a structure for employees that contains his salary, bonus and deductions. The program shall ask the user to enter these information for three employees ( Ahmed, Waleed and Amr). Then the program will print the total value shall be supplied by finance team.

### Expected Output

```
Please Enter Ahmed Salary: 1000
Please Enter Ahmed Bonus: 500
Please Enter Ahmed Deduction: 200
Please Enter Amr Salary: 2000
Please Enter Amr Bonus: 1000
Please Enter Amr Deduction: 0
Please Enter Waleed Salary: 3000
Please Enter Waleed Bonus: 350
Please Enter Waleed Deduction: 200
Total Value Needed is 7450
```

# Time To Code





## Structure and function

Structure can be passed by value to a function and also can be returned by function.

### Example

Function takes structure as an input argument

```
void function_2 (struct Employee x);
```

### Example

Function returns a structure

```
struct Employee function_1 (void);
```

## using typedef with structure

**typedef** can be used with structure. It would add the value of defining an object from structure by the structure name only without mentioning the word struct.

```
struct MyStruct
{
    u8    ID;
    u16   Salary;

};

typedef struct MyStruct Employee;
```

**Now**, we can create an object from Employee directly.

```
Employee Ahmed;
```

## Using typedef with structure

We can use **typedef** keyword at the moment of declaring the structure in the same statement

```
typedef struct MyStruct  
{  
    u8    ID;  
    u16   Salary;  
}Employee;
```

Name of the new type

**Now**, we can create an object from Employee directly.

```
Employee Ahmed;
```

Repeat Lab1 using the typedef keyword when declaring the structure.

### Expected Output

```
Please Enter Ahmed Salary: 1000
Please Enter Ahmed Bonus: 500
Please Enter Ahmed Deduction: 200
Please Enter Amr Salary: 2000
Please Enter Amr Bonus: 1000
Please Enter Amr Deduction: 0
Please Enter Waleed Salary: 3000
Please Enter Waleed Bonus: 350
Please Enter Waleed Deduction: 200
Total Value Needed is 7450
```

# Time To Code



## structure arithmetic

**Only** the **assignment operator** can be used with the structure, **to copy a content of structure to another structure.**

```
typedef struct  
{  
    u8 ID;  
    u8 Salary;  
}Employee;
```

```
Employee Ahmed = {100, 3000};  
Employee Ali;
```

```
Ali = Ahmed;
```

Copy the content of Ahmed to Ali



## structure arithmetic

Consider the following code

```
typedef struct
{
    u8 ID;
    u8 Salary;
}Employee;

Employee Ahmed = {100, 3000};
Employee Ali;

Ali = Ahmed;
```

The following examples shall give compilation error

```
if (Ali == Ahmed)
{
    /* Some Code */
}
```

Not Allowed

```
Ahmed += 20;
```

Not Allowed

```
int x = Ahmed + Ali;
```

Not Allowed

## Pointer to structure

```
typedef struct
{
    u8 ID;
    u8 Salary;
}Employee;

/* Create object from struct */
Employee Ahmed;

/* Create pointer to structure */
Employee *ptr = &Ahmed;

/* Accessing structure through pointer */
*ptr.ID = 10;
ptr -> ID = 10; ← Using the arrow operator with structure
```

### Note

The arrow operator `->` is used with pointer to struct as replacement to the dereference operator `*` and the dot operator `.`

## Array of structure

```
typedef struct
{
    u8 ID;
    u16 Salary;
}Employee;

void main(void)
{
    u8 i;

    /* Create an array of sturcute Empolyee */
    Employee Arr[10];

    /* Loop to fill the array */
    for (i = 0; i<10; i++)
    {
        Arr[i].ID    = i;
        Arr[Salary] = 1000 * i;
    }
}
```

### Note

This code fills 10 elements array with ID equals to the loop iteration counter and salary equals to ID multiplied by 1000



Write a C code to manage a class of 10 students. Each student studies 4 subjects Math, Language, Physics and Chemistry.

First define an array of 10 elements and assign random grades for students. The system will ask the user to enter the student ID then the system will show its grades. The software shall manage wrong IDs.

# Time To Code

### Expected Output

```
Please Enter Student ID: 5  
Math Grade: 70  
Language Grade: 80  
Math Grade: 75  
Math Grade: 84
```

```
Please Enter Student ID: 70  
Student ID is not correct
```



## Bit Field

Bit field is a member of a **struct** that has size of certain number of bits

### Syntax

**type** **name** : **number\_of\_bits** ;

### Example

```
typedef struct
{
    u8 x : 4 ;
    u8 y : 2 ;
    u8 z : 1 ;
}mystruct;
```

colon

This line creates a **u8** variable, and make the first **4** bits **x**, the remaining bits are save for later use

This line use **2** bits from the remaining **4** bits and name them **y**

This line use **1** bit from the remaining **2** bits and name it **z**

### Note:

The total size of this struct is **1 byte**, only **7** of them are used as a bit fields, and the last bit is not used

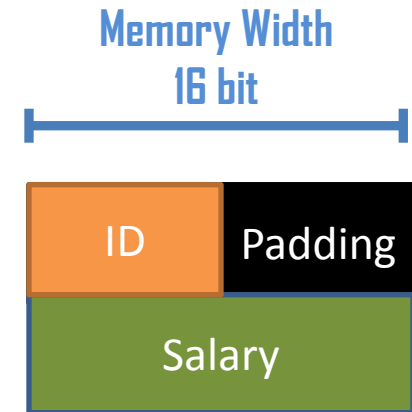
## Size of structure

*Size of structure* is **not always** equals to summation of the members size !

To understand how the size of the structure is calculated, we need to understand how the structure is saved in the memory. Assume the following example:

```
struct Employee
{
    u8    ID    ;
    u16   Salary;
} Ahmed;
```

Assuming **16 bit** Memory Width



After the compiler allocates **8 bits** for the ID member, it shall allocate the next member Salary which is **16 bit**. The compiler used only 8 bits of the first location, So it can break down the member salary and save **half of it in the first location** and the other **half in the second location**. But with this scenario the member Salary would be read in **2 cycles instead of 1 cycle**. For that, the compiler decide to Pad (Not use) the remaining part of first location and save the Salary in the second location. So This structure is **4 Byte**.

## Size of structure

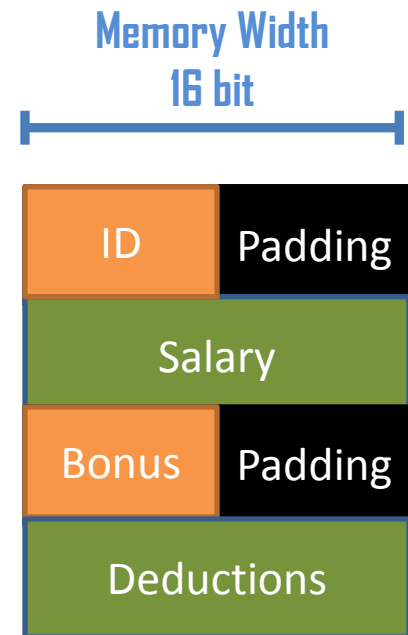
Assuming 16 bit memory width, can you expect what would be the size of this structure ... ?

```
struct Employee
{
    u8    ID           ;
    u16   Salary       ;
    u8    Bonus        ;
    u16   Deductions   ;
} Ahmed;
```

## Size of structure

Assuming 16 bit memory width, can you expect what would be the size of this structure ... ?

```
struct Employee
{
    u8    ID           ;
    u16   Salary       ;
    u8    Bonus        ;
    u16   Deductions   ;
} Ahmed;
```



This structure is **8 Bytes**

## Size of structure

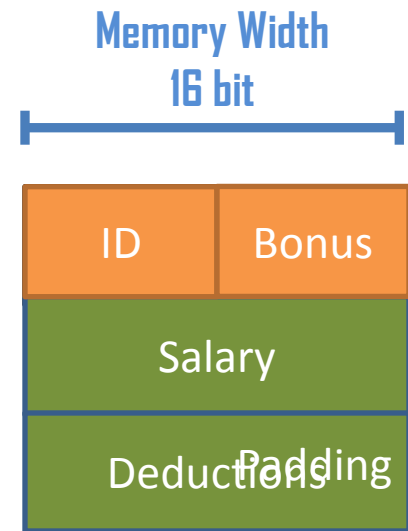
What if we re-arranged the members like that, can you expect the size of the structure ... ?

```
struct Employee
{
    u8    ID           ;
    u8    Bonus        ;
    u16   Salary       ;
    u16   Deductions   ;
}Ahmed;
```

## Size of structure

Assuming 16 bit memory width, can you expect what would be the size of this structure ... ?

```
struct Employee
{
    u8    ID          ;
    u8    Bonus       ;
    u16   Salary      ;
    u16   Deductions  ;
} Ahmed;
```

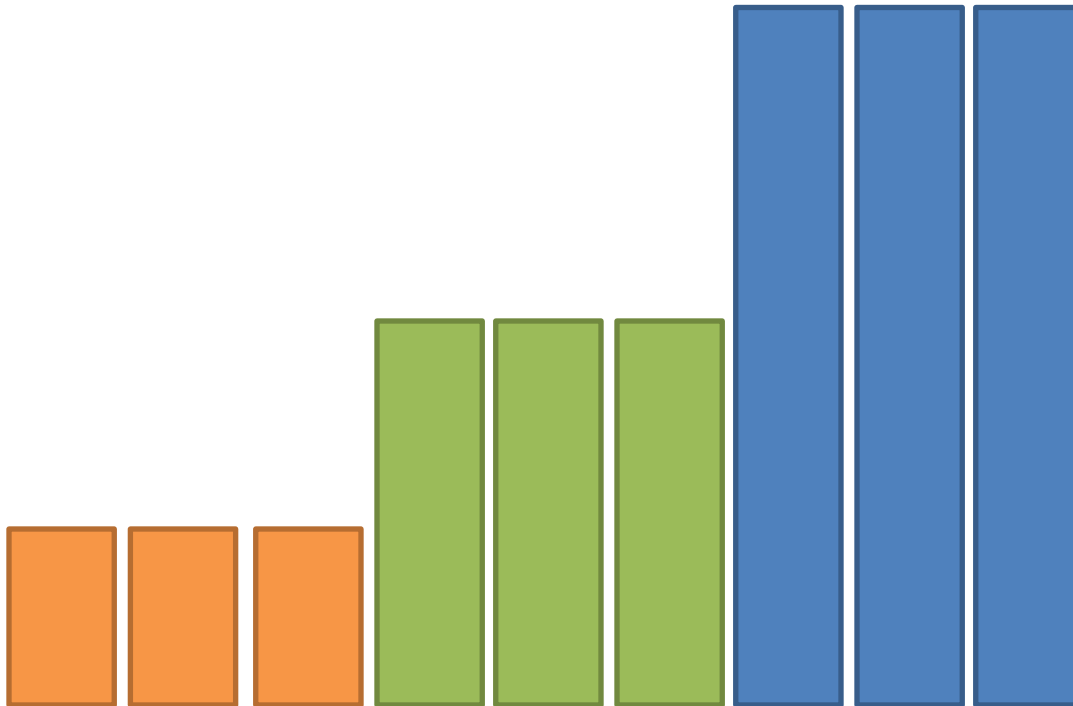


This structure is **6 Bytes**

## Size of structure

From the previous example, we can conclude that arranging the structure members can affect its size.

The general rule states that always arrange the members in **ascending** order or **descending** order to get the lowest size.





Could you expect the size of this struct ... ?  
Write a C code to print the size of this structure  
and verify that it meets your expectations.

After that modify the structure to achieve the a  
minimum size. Can you see the difference ... ?

**Note that:**

Your tool chain consider that the memory width  
is 4 byte.

Expected Output

```
typedef struct
{
    u16    x;
    u32    y;
    u16    z;
    u32    k;
}mystruct;
```

# Time To Code



## union data type

Like struct, this data type contains many types under the same umbrella. But only one of them is valid at certain time. Because all members share the same memory locations. So, the size of the union is the size of the biggest member

### Syntax

```
union union_name
{
    member_1_Type member_1_Name ;
    member_2_Type member_2_Name;
    |
    |
};
```



### Note

*union name is optional*

## union Example

This example creates a union named **myunion**. This union has three members one called **x, y and z**. The size of any object created from the union is **4** byte (Size of z).

```
union myunion
{
    u8    x;
    u16   y;
    u32   z;
};
```

### Note

**Because x, y and z are sharing the same location, then writing to any of them will corrupt the other members.**

## union use case

```
typedef union  
{
```

```
    struct
```

```
    {
```

```
        u8 B0 : 1;
```

```
        u8 B1 : 1;
```

```
        u8 B2 : 1;
```

```
        u8 B3 : 1;
```

```
        u8 B4 : 1;
```

```
        u8 B5 : 1;
```

```
        u8 B6 : 1;
```

```
        u8 B7 : 1;
```

```
    }Bit;
```

```
    u8 Byte;
```

```
}Register;
```

Defining a struct with no name

The struct contains 8 bit fields each one of them is 1 bit. So the total size of this struct is 8 bit

Creating an object of the struct, the object is called bit

u8 object sharing the same memory with the struct

## union use case clarification

توضيح

The size of this union which called Register is 8 bit as it is contains two members **struct** and **u8** and both of them is 8 bit size.

When you create an object of this union, you can either access the **whole variable** at time or access the variable **bit by bit**. This technique is widely used in Embedded systems development

```
typedef union
{
    struct
    {
        u8 B0 : 1;
        u8 B1 : 1;
        u8 B2 : 1;
        u8 B3 : 1;
        u8 B4 : 1;
        u8 B5 : 1;
        u8 B6 : 1;
        u8 B7 : 1;
    }Bit;

    u8 Byte;
}Register;
```

### Example

```
/* Defining a variable called x of type Register */
Register x;

/* Acces the whole variable */
x.Byte = 34;

/* Access the first bit of the variable */
x.Bit.B0 = 1;
```

## enum data type

**enum** is a user defined data type consisting of a set of named constants called **enumerators**. The enumerators are a set of integer constants represented by identifiers.

العناوين

### Syntax

```
enum enum_name  
{  
    enum-list  
};
```

Or

### Syntax

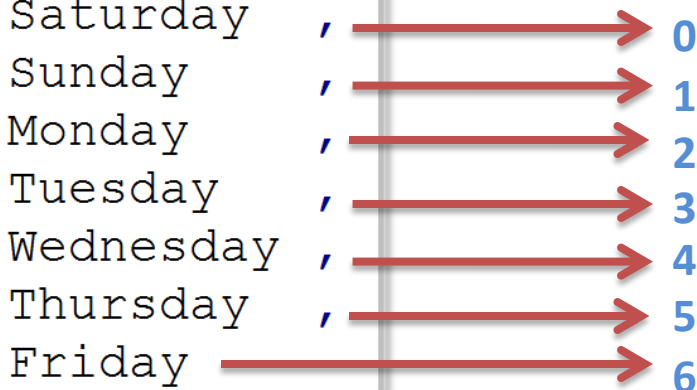
```
typedef enum optional_enum_name  
{  
    enum-list  
}enum_type_name;
```

### enum rules:

- 1- First enumerator value is **0** if it is not directly defined
- 2- Each enumerator value equals to the **preceding enumerator incremented by 1** if it is not directly defined.
- 3- May enumerators in the same enum **may have the same value !**

## enum Examples

```
enum Days
{
    Saturday ,
    Sunday ,
    Monday ,
    Tuesday ,
    Wednesday ,
    Thursday ,
    Friday
};
```




This is  
example 1

```
/* Creating variable x of type enum Days */
enum Days x;
```

```
/* Assign value to x from the list of enumerators */
x = Sunday; /* It means that x = 1 */
```

## enum Examples

```
enum Days
{
    Saturday = 5 ,
    Sunday    = 6 ,
    Monday    = 7 ,
    Tuesday   = 4 ,
    Wednesday = 5 ,
    Thursday  = 6 ,
    Friday    = 7
};
```



This is  
example 2

```
/* Creating variable x of type enum Days */
```

```
enum Days x;
```

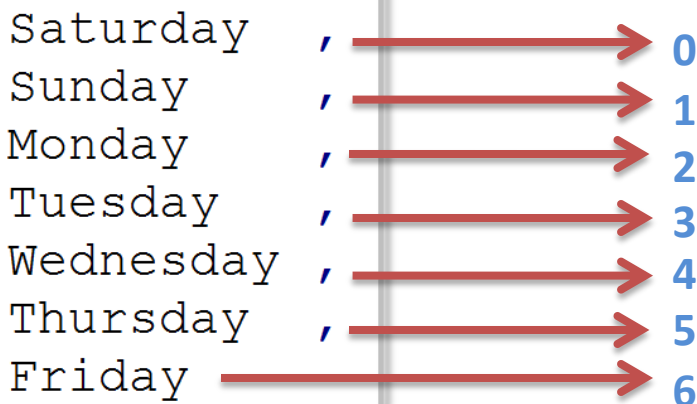
```
/* Assign value to x from the list of enumerators */
```

```
x = Friday; /* It means that x = 7 */
```



## enum Examples

```
enum Days
{
    Saturday ,
    Sunday ,
    Monday ,
    Tuesday ,
    Wednesday ,
    Thursday ,
    Friday
};
```



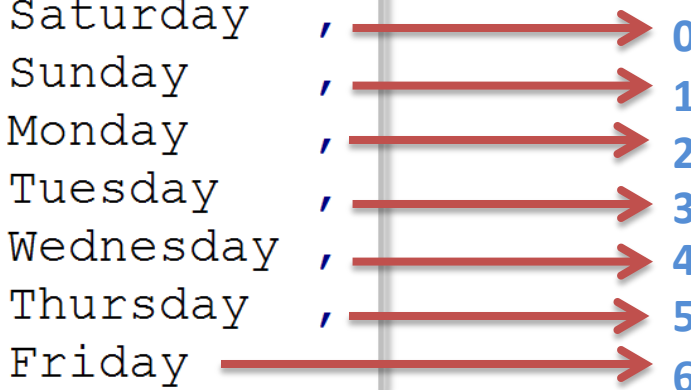
This is  
example 3

```
/* Creating variable x of type enum Days */
enum Days x;
```

```
/* You can assign any value even if it is out of enumerator list */
x = 10;
```

## enum Examples

```
enum Days
{
    Saturday ,
    Sunday ,
    Monday ,
    Tuesday ,
    Wednesday ,
    Thursday ,
    Friday
};
```



This is  
example 4

```
/* Creating variable x of type integer */
int x;
```

```
/* You can use the enumerators with any type even if it is not enum */
x = Saturday; /* It means that x = 0 */
```

## enum Summary

- ❑ First enumerator value is **0** if it is not directly defined
- ❑ Each enumerator value equals to the **preceding** السابق **enumerator incremented by 1** if it is not directly defined.
- ❑ May enumerators in the same enum **may have the same value !**
- ❑ **Size of enum** variable is the same as the size of int variable
- ❑ enum variable can be equals to a value of the enumerator list or **any other value**
- ❑ Any data type can use the enumerator list

The End ...





[www.imtschool.com](http://www.imtschool.com)



[www.facebook.com/imaketechologyschool/](http://www.facebook.com/imaketechologyschool/)

*This material is developed by IMTSchool for educational use only  
All copyrights are reserved*