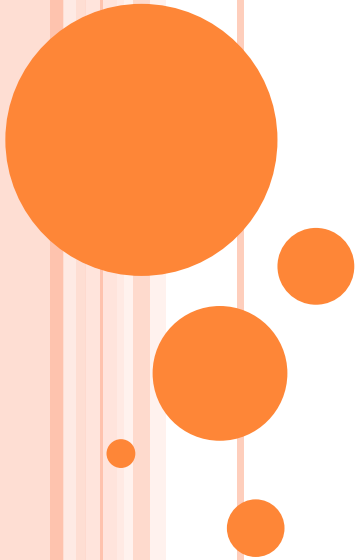


Structures

STRUCTURES

- A struct (or structure) is a collection of variables (can be of different types) under a single name.
- C structures are special, large variables which contain several named variables inside. Structures are the basic foundation for objects and classes in C. Structures are used for:
  - Serialization of data
  - Passing multiple arguments in and out of functions through a single argument
  - Data structures such as linked lists, binary trees, and more
- **structures** to store data of different types.
- For example, you are a student. Your name is a string and your phone number and roll_no are integers. So, here name, address and phone number are those different types of data.

How to define structures?

- Before you can create structure variables, you need to define its data type.
- To define a struct, the *struct* keyword is used.

Syntax of struct

```
struct structureName
{
    dataType member1;
    dataType member2; ...
};
```

DECLARATION/DEFINITION OF A STRUCTURE

- **Syntax of struct**

```
struct tagname{
        char    x;
        int     y;
        float   z;
};
```

- tagname are data types names int , char, etc.

struct tagname structvariable;

EXAMPLE

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

- Here, a derived type ***struct Person*** is defined. Now, you can create variables of this type.
- When a struct type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables

CREATE STRUCT VARIABLES

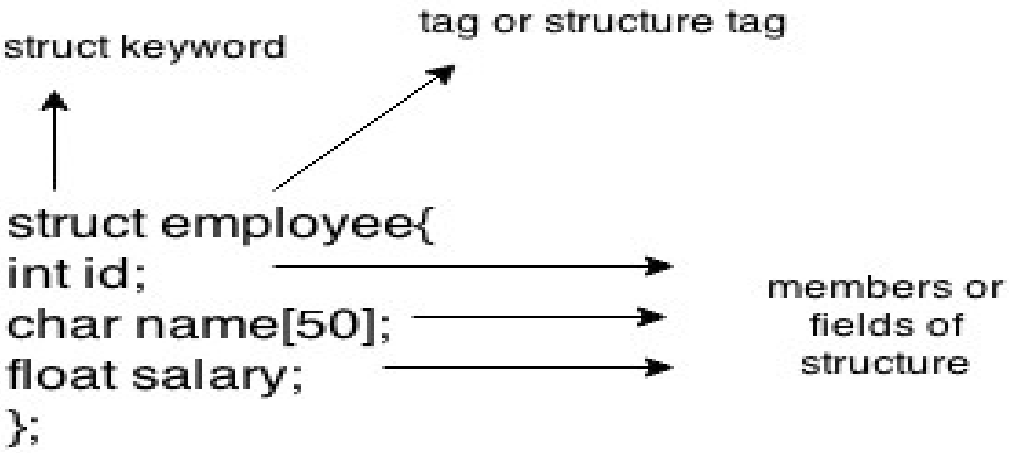
- Here's how we create structure variables:

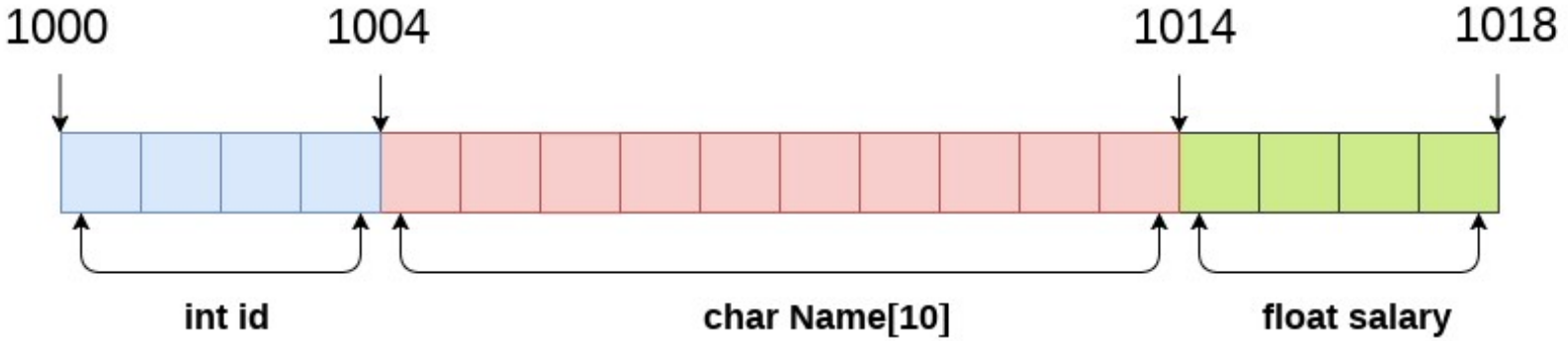
```
struct Person
 {
 char name[50];
 int citNo;
 float salary;
};
 int main()
 {
struct Person person1, person2, p[20];
 return 0;
 }
```

ANOTHER WAY OF CREATING A STRUCT VARIABLE

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

- In both cases, **two variables person1, person2, and an array variable p having 20 elements of type struct Person** are created.
- **Which approach is good ?**
- If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.
- If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function

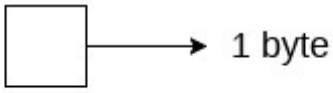





```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

sizeof (emp) = 4 + 10 + 4 = 18 bytes

where;
sizeof (int) = 4 byte
sizeof (char) = 1 byte
sizeof (float) = 4 byte



```
// A variable declaration with structure declaration.
struct Point
{
   int x, y;
} p1;  // The variable p1 is declared with 'Point'


// A variable declaration like basic data types
struct Point
{
   int x, y;
};

int main()
{
   struct Point p1;  // The variable p1 is declared like a
   normal                                    variable
}
```

UNNAMED STRUCTURE

- An anonymous struct declaration is a declaration that declares neither a tag for the struct, nor an object or typedef name.
- As they have no names, so we cannot create direct objects of it. We use it as nested structures.

Examples

```
struct
{
datatype variable;
... };
```

- In this example we are making one structure, called point, it is holding an anonymous structure. This is holding two values x, y. We can access the anonymous structure directly.

How to initialize structure members?

- Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

```
struct Point
{
  int x = 0;  // COMPILER ERROR:  cannot initialize
              members here
  int y = 0;  // COMPILER ERROR:  cannot initialize
              members here
};
```

- The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

How to initialize structure members?

- Structure members can be initialized using curly braces '{}'. For example, following is a valid initialization.

```
struct Point
{
  int x, y;
};
int main()
{
  // A valid initialization. member x gets value 0 and y
 // gets value 1.  The order of declaration is followed.
  struct Point p1 = {0, 1};
}
```

ACCESS MEMBERS OF A STRUCTURE

- There are two types of operators used for accessing members of a structure.
- . - Member operator
- -> - Structure pointer operator
- Suppose, you want to access the salary of person2. Here's how you can do it.
- person2.salary

ACCESS MEMBERS OF A STRUCTURE VARIABLE & POINTER

- Many variables are created inside the structure, to access them and do some operations:
- **structvariable.x='A'**

/* this will write 'A' for the element x of structure structvariable*/

- structvaraible.y=20;
- structvariable.z=10.20f;

- The objects to a structure can also be a pointer

structtagname *structPtrVariable;

- To access the elements inside the structure we should be using the following syntax
- structPtrVariable->x = 'A' // here '.' is replace by '->' structPtrVariable->y = 20; structPtrVariable->z = 10.20f;

ALWAYS USE STRCPY TO ASSIGN A STRING VALUE TO A STRING VARIABLE.

- The reason for this is that in C, we cannot equate two strings (i.e. character arrays). If we had written ' p1.name="Brown"; ', that would have given an error.
- Therefore, by writing strcpy(p1,"Brown"), we are copying the string 'Brown' to the string variable p1.name.
- struct student p1 = {1,"Brown",123443}; → This is also one of the ways in which we can initialize a structure. In next line, we are just giving values to the variables and printing it.

SIZEOF

- The sizeof for a struct is not always equal to the sum of sizeof of each individual member.
- This is because of the padding added by the compiler to avoid alignment issues.
- Padding is only added when a structure member is followed by a member with a larger size or at the end of the structure.
- Different compilers might have different alignment constraints as C standards state that alignment of structure totally depends on the implementation.

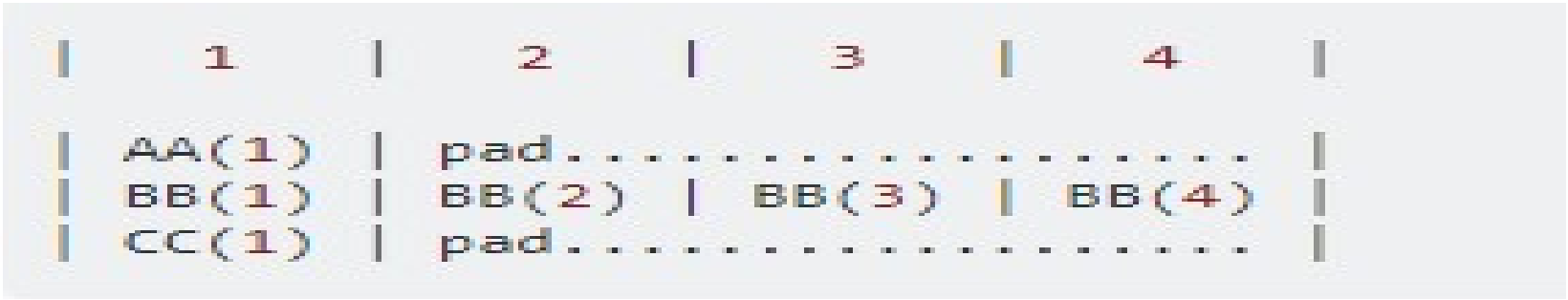
SIZEOF

- #pragma pack instructs the compiler to pack structure members with particular alignment.
- Most compilers, when you declare a struct, will insert padding between members to ensure that they are aligned to appropriate addresses in memory (usually a multiple of the type's size).
- This avoids the performance penalty (or outright error) on some architectures associated with accessing variables that are not aligned properly.

```
struct Test
{
    char AA;
    int BB;
    char CC;
};
```

SIZEOF

- The compiler could choose to lay the struct out in memory like this:



- and sizeof(Test) would be 4 × 3 = 12, even though it only contains 6 bytes of data.
- The most common use case for the #pragma (to my knowledge) is when working with hardware devices where you need to ensure that the compiler does not insert padding into the data and each member follows the previous one.

SIZEOF

- With #pragma pack(1), the struct above would be laid out like this:

```
|   1   |

| AA(1) |
| BB(1) |
| BB(2) |
| BB(3) |
| BB(4) |
| CC(1) |
```

- And sizeof(Test) would be 1 × 6 = 6.
- With #pragma pack(2), the struct above would be laid out like this:

```
struct Test
{
    char AA;
    char CC;
    int BB;
};
```

- and with #pragma pack(2), the struct would be laid out like this:
- and sizeOf(Test) would be 3 × 2 = 6.

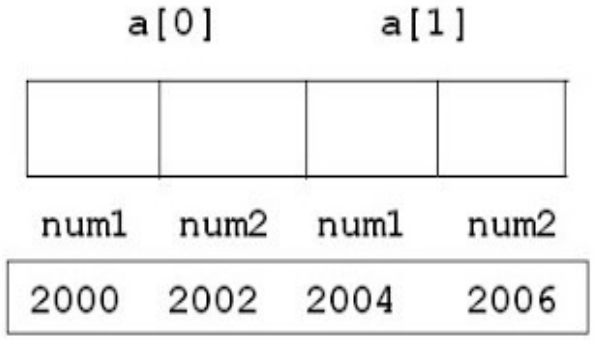
```
|   1   |   2   |

| AA(1) | CC(1) |
| BB(1) | BB(2) |
| BB(3) | BB(4) |
```

WITHOUT USING SIZEOF OPERATOR.

- Sizeof operator is used to evaluate the size of any data type or any variable in c programming. Using sizeof operator is straight forward way of calculating size but following program will explain you how to calculate size of structure without using sizeof operator.
- Steps to Calculate Size of Structure :
  - Create Structure .
  - Create an array of Structure ,
    Here a[2].
  - Individual Structure Element and Its Address –

```
Address of a[0].num1 = 2000
Address of a[0].num2 = 2002
Address of a[1].num1 = 2004
Address of a[1].num2 = 2005
```



1. **&a[1].num1 –** 2004
2. **&a[0].num1 –** 2000
3. Difference Between Them = 2004 – 2000 = 4 Bytes (Size of Structure)

Passing Structure to Function in C Programming

- A structure can be passed to any function from main function or from any sub function.
- Structure definition will be available within the function only.
- It won't be available to other functions unless it is passed to those functions by value or by address(reference).
- Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program.
- Structure can be passed to function as a Parameter.
- function can also Structure as return type.

POINTERS TO STRUCTURES

- Like we have pointers for int, char and other data-types, we also have pointers pointing to structures. These pointers are called **structure pointers.**

```
struct structure_name
{
  data-type member-1;
  data-type member-1;
  data-type member-1;
  data-type member-1;
};
main()
{
  struct structure_name *ptr
}
```

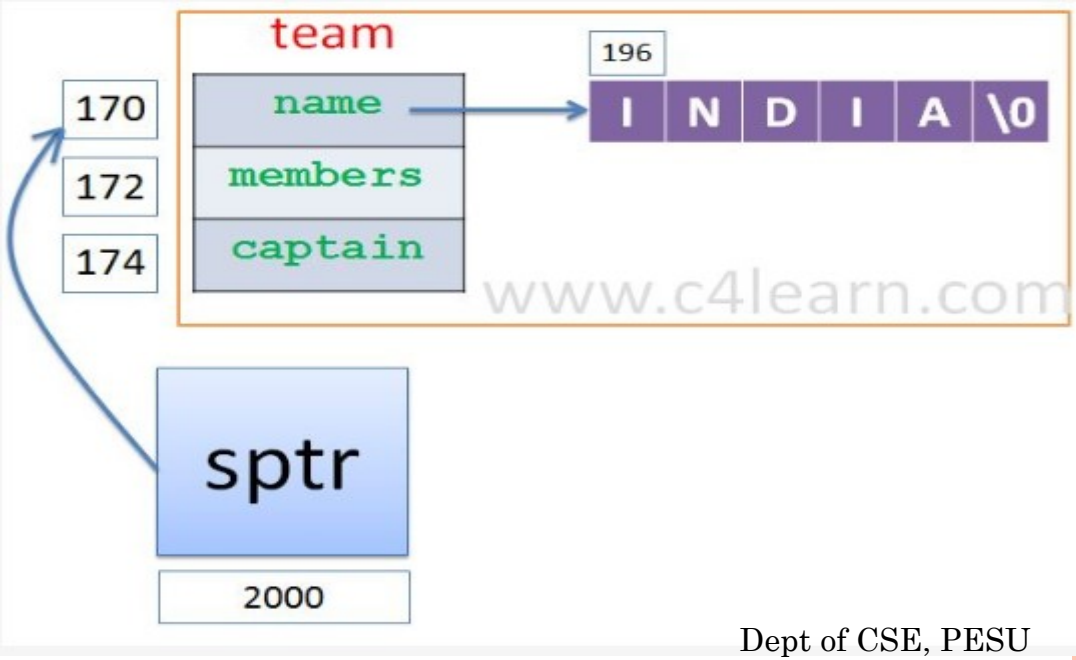
- Address of Pointer variable can be obtained using '&' operator.
- Address of such Structure can be assigned to the Pointer variable .
- Pointer Variable which stores the address of Structure must be declared as Pointer to Structure .

POINTERS TO STRUCTURES

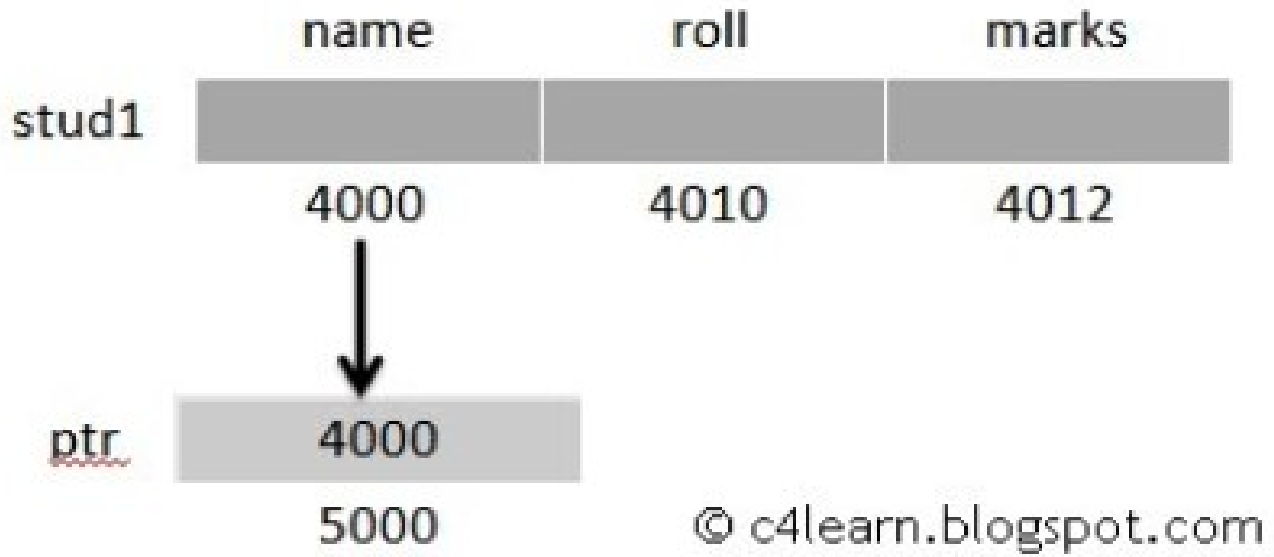
- Pointer which stores address of structure is called as “Pointer to Structures.

  Explanation :

- sptr is pointer to structure address.
- -> and (*). both represents the same.
- These operators are used to access data member of structure by using structure’s pointer.



```
struct student_database
{
    char name[10];
    int roll;
    int marks;
}stud1;
struct student_database *ptr;
ptr = &stud1;
```

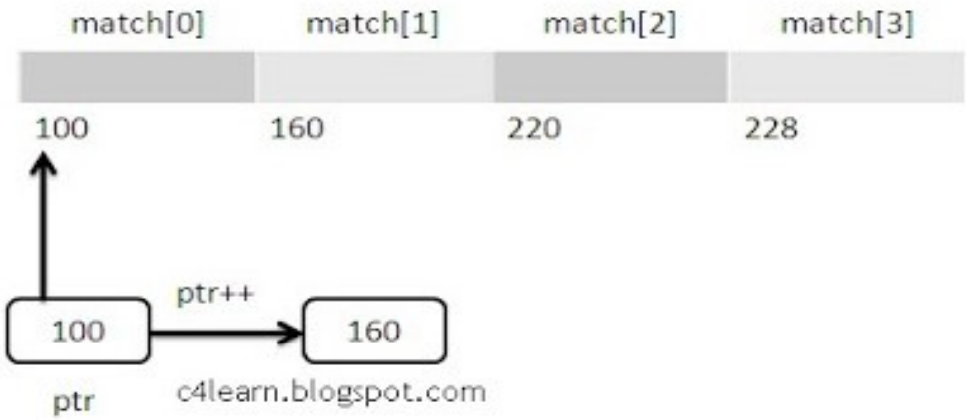


```
#include <stdio.h>
int main()
{
  struct student
  {
    char name[30];
    int roll_no;
  };
  struct student stud = {"sam",1};
  struct student *ptr;
  ptr = &stud;
  printf("%s %d\n",stud.name,stud.roll_no);
  printf("%s %d\n",ptr->name,ptr->roll_no);
  return 0;
}
```

- ptr = &stud; → We are making our pointer 'ptr' to point to structure 'stud'.
- This was the same thing that we have done earlier up till here. Now, coming to printf.
- printf("%s %d\n",ptr->name,ptr->roll_no); → Yes, we use -> to access a structure from its pointer.
- It is same as we were using (.) with structure, we use -> with pointer.

Pointer to Array of Structure

- Pointer Variable can also Store the Address of the Structure Variable.
- Pointer to Array of Structure stores the Base address of the Structure array.
- Suppose




```
struct Cricket {
        char team1[20];
        char team2[20];
        char ground[18];
        int result;}match[4]={{"IND","AUS","PUNE",1},
                              {"IND","PAK","NAGPUR",1},
                              {"IND","NZ","MUMBAI",0},
                              {"IND","SA","DELHI",1} };
```

Pointer Is Declared and initialized as —

- struct Cricket *ptr = match
- Here the address of match[0] is stored in pointer Variable ptr.

Expression	Meaning
++ptr->length	Increment the value of length
(++ptr)->length	Increment ptr before accessing length
(ptr++)->length	Increment ptr after accessing length
*ptr->name	Fetch Content of name
*ptr->name++	Incrementing ptr after Fetching the value
(*ptr->name)++	Increments whatever str points to
*ptr++->name	Incrementing ptr after accessing whatever str points to

DESIGNATED INITIALIZATION

- Designated Initialization allows structure members to be initialized in any order. This feature has been added in C99 Standard.

```
#include<stdio.h>
struct Point
{
  int x, y, z;
};

int main()
{
  // Examples of initialization using designated initialization
  struct Point p1 = {.y = 0, .z = 1, .x = 2};
  struct Point p2 = {.x = 20};

  printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
  printf ("x = %d", p2.x);
  return 0;
}
```

NESTED STRUCTURES: STRUCTURE WITHIN STRUCTURE

- Structure is a user define data type that contains one or more different type of variables.
- When a structure definition has an object of another structure, it is known as Nested Structure.
- Structure written inside another structure is called as nesting of two structures.

NESTED STRUCTURES :WAY 1 : DECLARE TWO SEPARATE STRUCTURES

```
struct date
{
  int date;
  int month;
  int year;
};
struct Employee
  {
  char ename[20];
  int ssn;
  float salary;
  struct date doj;
}emp1;
```

- Accessing Nested Elements :
  - Structure members are accessed using dot operator.
  - ‘date‘ structure is nested within Employee Structure.
  - Members of the ‘date‘ can be accessed using ’employee’
  - emp1 & doj are two structure names (Variables)

NESTED STRUCTURES :WAY 1 : DECLARE TWO SEPARATE STRUCTURES

```
struct date
{
  int date;
  int month;
  int year;
};
struct Employee
  {
  char ename[20];
  int ssn;
  float salary;
  struct date doj;
}emp1;
```

- Explanation Of Nested Structure :
  - Accessing Month Field : emp1.doj.month
  - Accessing day Field : emp1.doj.day
  - Accessing year Field : emp1.doj.year

Nested Structures : Way 2 : Declare Embedded structures

```
struct Employee
{
   char ename[20];
   int ssn;
   float salary;
   struct date
      {
      int date;
      int month;
      int year;
      }doj;
}emp1;
```

- Accessing Nested Members :
- Accessing Month Field : emp1.doj.month
- Accessing day Field : emp1.doj.day
- Accessing year Field : emp1.doj.year

NESTED STRUCTURE INITIALIZATION

- Structure can be initialized by two methods

Method 1

- struct studentDetails std={"Mike",21,{15,10,1990}};

OR

- struct studentDetails std={"Mike",21,15,10,1990};
- Here, "Mike" and 21 will be stored in the members name and age of structure studentDetails, while values 15, 10 and 1990 will be stored in the dd, mm, and yy of structure dateOfBirth.

Method 2

- struct dateOfBirth dob={15,10,1990};
- struct studentDetails std={"Mike",21,dob};
- Initialize the first structure first, then use structure variable in the initialization statement of main (parent) structure.

KEYWORD TYPEDEF

- We use the *typedef* keyword to create an alias name for data types.
- It is commonly used with structures to simplify the syntax of declaring variables.

```
struct Distance{
     int feet;
     float inch;
};

int main() {
     structure Distance d1, d2;
}
```

```
typedef struct Distance{
      int feet;
      float inch;
} distances;

int main() {
      distances d1, d2;
}
```

KEYWORD TYPEDEF

- It allows us to introduce synonyms for data types which could have been declared some other way.
- It is used to give New name to the Structure.
- New name is used for Creating instances, Passing values to function, declaration etc…

typedef	#define
typedef is limited to giving symbolic names to types only	#define can be used to define an alias for values as well, e.g., you can define 1 as ONE, 3.14 as PI, etc
typedef interpretation is performed by the compiler	#define statements are performed by preprocessor.
typedef should be terminated with semicolon	#define should not be terminated with a semicolon
typedef is the actual definition of a new type	#define will just copy-paste the definition values at the point of use
typedef follows the scope rule which means if a new type is defined in a scope (inside a function), then the new type name will only be visible till the scope is there	#define, when preprocessor encounters #define, it replaces all the occurrences, after that (No scope rule is followed).

ARRAY OF STRUCTURES

- Structure is used to store the information of One particular object but if we need to store such 100 objects then Array of Structure is used.
- We can also declare an array of structure variables. in which each element of the array will represent a structure variable.

Example : struct employee emp[5];

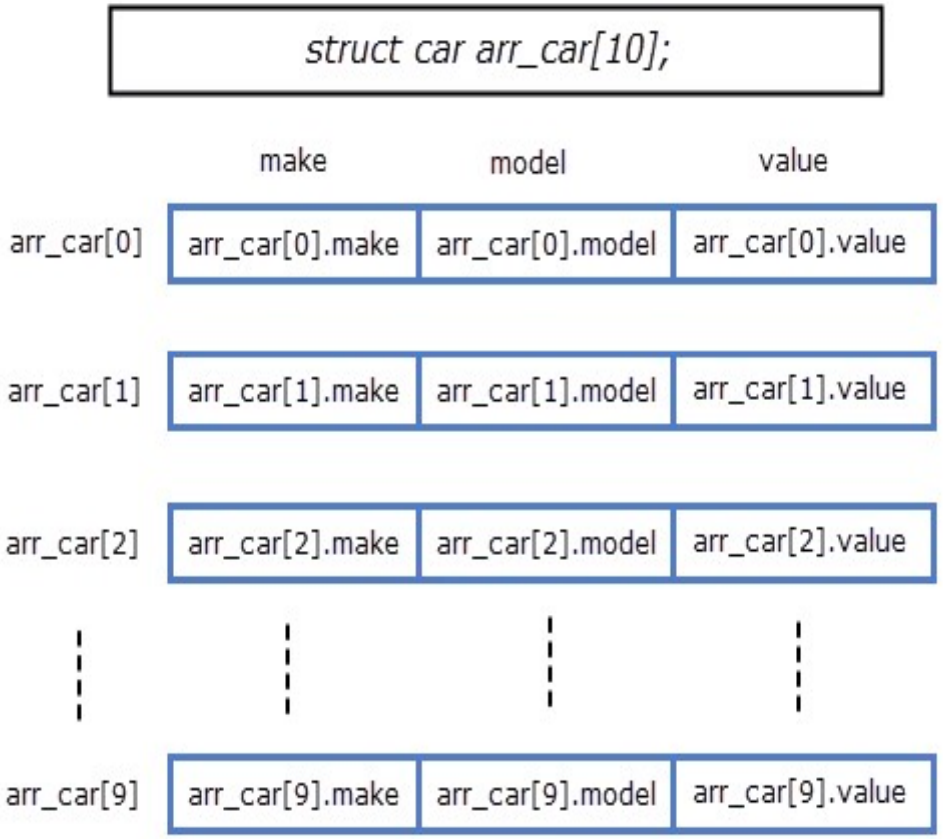
- **Accessing Element in Structure Array**
  - Array of Structure can be accessed using dot[.] operator.
  - Here Records of 3 Employee are Stored.
  - 'for loop' is used to Enter the Record of first Employee.
  - Similarly 'for Loop' is used to Display Record.

Array of Structures in C

```
struct car
{
    char make[20];
    char model[30];
    int year;
};
```

Here we can declare an array of structure car.

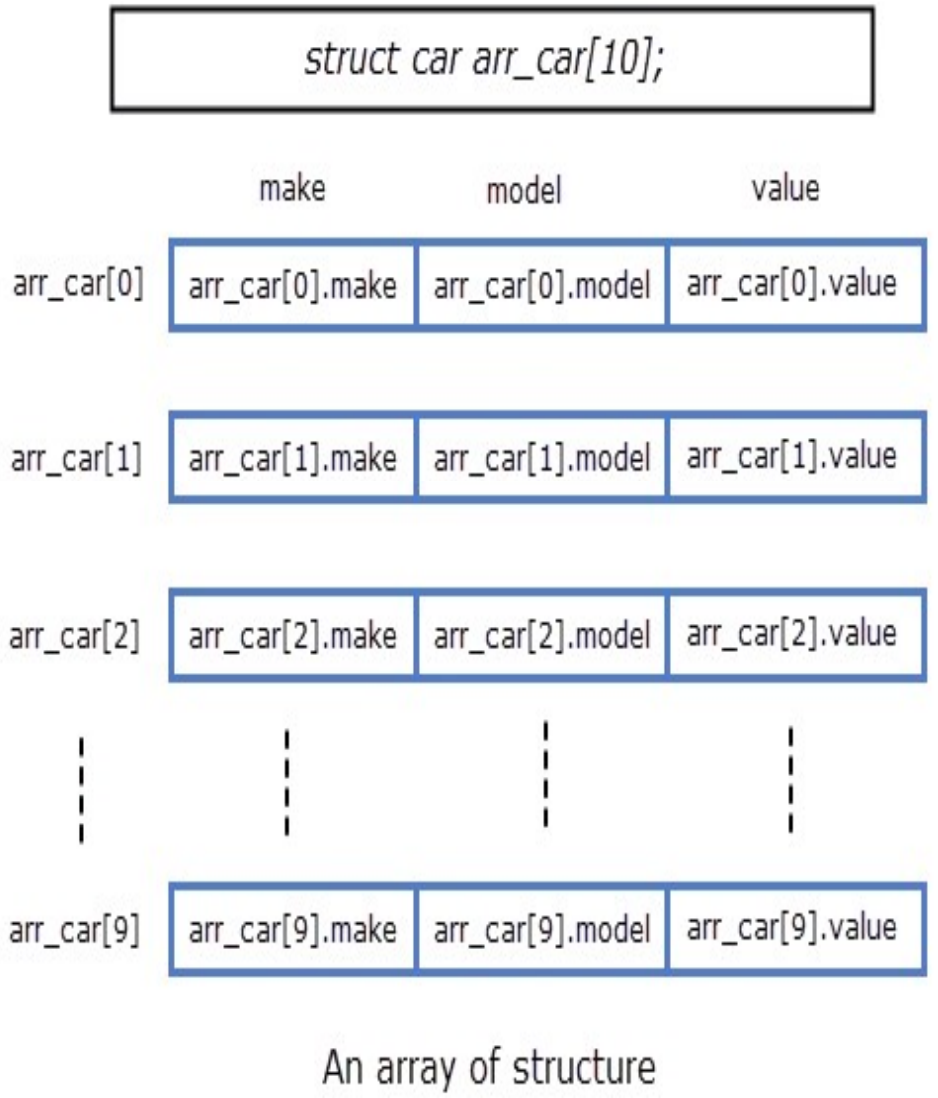
```
struct car arr_car[10];
```



An array of structure

Array of Structures in C

- Here arr_car is an array of 10 elements where each element is of type struct car.
- We can use arr_car to store 10 structure variables of type struct car.
- To access individual elements we will use subscript notation ([]) and to access the members of each element we will use dot (.) operator as usual.
  - arr_stu[0] : points to the 0th element of the array.
  - arr_stu[1] : points to the 1st element of the array.
  - arr_stu[0].name : refers to the name member of the 0th element of the array.
  - arr_stu[0].roll_no : refers to the roll_no member of the 0th element of the array.
  - arr_stu[0].marks : refers to the marks member of the 0th element of the array.



An array of structure

Operations on struct variables in C

- Only one operation can be performed for struct. The operation is assignment operation.
- Any other operation (e.g. equality check) is not allowed on *struct* variables.

PROGRAMS:

- WAP that asks the user to enter name, roll no and marks in 2 subjects and calculates the average marks of each student

Using normal variable	Using pointer variable

Syntax:
struct tag_name
{
data type var_name1;
data type var_name2;
data type var_name3;
};
	Syntax:
	struct tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};

Example:
struct student
{
int mark;
char name[10];
float average;
};
	Example:
	struct student
	{
	int mark;
	char name[10];
	float average;
	};

Using normal variable	Using pointer variable
Declaring structure using normal variable: struct student report;	**Declaring structure using pointer variable:** struct student *report, rep;
Initializing structure using normal variable: struct student report = {100, “Mani”, 99.5};	**Initializing structure using pointer variable:** struct student rep = {100, “Mani”, 99.5}; report = &rep;
Accessing structure members using normal variable: report.mark; report.name; report.average;	**Accessing structure members using pointer variable:** report -> mark; report -> name; report -> average;

Important Notes for Declaring Structure Variable

- Closing brace of structure type declaration must be followed by semicolon.
- struct date { int date; char month[20]; int year; }; Don't forgot to use 'struct' keyword
- struct date Memory will not be allocated just by Creating instance or by declaring structure
- Memory will not be allocated just by writing following declaration –
- struct date { int date; char month[20]; int year; }; Memory will be allocated on defining it i.e –
- date d1; Generally Structures are written in Global Declaration Section , But they can be written inside main.
- int main() { struct student_database { char name[10]; int roll; int marks; }stud1; return(0); } It is not necessary to initialize all members of structure.
- struct student_database { char name[10]; int roll; int marks; }stud1 = {"Pritesh"}; For Larger program , Structures may be embedded in separate header file.

USES OF STRUCTURES IN C

- C Structures can be used to store huge data. Structures act as a database.
- C Structures can be used to send data to the printer.
- C Structures can interact with keyboard and mouse to store the data.
- C Structures can be used in drawing and floppy formatting.
- C Structures can be used to clear output screen contents.
- C Structures can be used to check computer's memory size etc.
- We cannot use operators like +,- etc. on Structure variables

USES OF STRUCTURES IN C

- No Data Hiding: C Structures do not permit data hiding. Structure members can be accessed by any function, anywhere in the scope of the Structure.
- Functions inside Structure: C structures do not permit functions inside Structure.
- Static Members: C Structures cannot have static members inside their body.
- Access Modifiers: C Programming language do not support access modifiers. So they cannot be used in C Structures.
- Construction creation in Structure: Structures in C cannot have constructor inside Structures.