పింగళి వెంకయ్య నమః

C programming for problem solving.

UNIT-IV Syllabus.

Userdefined data types: Structure and Unions

Initialization, accessing structures, operations on structures, complex structures, Nested structures, structures containing arrays, structures Containing pointers, arrays of structures, structures and functions passing structures through pointers, self-referential structures. unions, bitfields, C programming examples, Commandline arguments, preprocessor commands.

Structures in C:

First of all, Why we need a variable?

If you want to store a value in computer memory, then we need a variable.

Let us say I want to store an integer value i.e. 5.

We will declare a variable like: int a=5; → value.
→ variable name.
→ datatype.

→ If you want to store suppose 60 values like roll numbers of 60 students, we use arrays and we know why we need arrays.

int a[60]; ⇒ This is declaration of an array and we can store any integer value of 60 values or 60 integer values.

(or) we can declare as: int rollno[60]; // so roll numbers of 60 students we can store under a single variable name i.e. why we need arrays.

Arrays are collection of homogeneous data items.

↳ we can store more than one data item of the same data type

→ But in real life, we have to process the data of different different datatypes like what?

Suppose I want to store the information of the student, like "Roll Number", "Name", "marks", "Address" and so on---

→ for simplicity, we take rollno, name, marks.

RollNO will be of type integer.
Name will be of type String
Marks will be of type float

→ we have 3 types (datatypes)
→ we have different, different types of information about a student.

→ like integer, string, float or may be if we consider address it can be of integer/string type or we can have much more information about a student.

→ Suppose in a class, there are 60 students.

→ For Example we take/ consider two students

Student 1 data	Student 2 data.
For student 1 i will take different different variables like.	for student 2 also we will take different different variables.
int rollno$1; char names1[30]; float marks$2;	int rollnos2; char names2[30]; float marks$2;

For storing information about two students we have taken 3 and 3 i.e total 6 variables.

Suppose we have 60 students, we can imagine how many variables we have to take.

→ And in information we have only 3 things are considered here.

→ Suppose you are developing a software, obviously for the student information, for may be a University, you want to develop the software, to store the information or to store information of faculty also, so on---.

→ So How many variables we will take-)

Obviously it is not possible to take variables to store the data.

vvv imp may be we can take arrays-)

But arrays, yes we can store roll no's of 60 students like more than 1 student, but that data should be of same type. (Homogeneous).

int rollno[60]; → this is an array and i can store only 60 integer values.

float names[60]; → Here i can store marks of 60 students but that will be of float type only.

→ so here also we have to take different different arrays.

one array for name, one for rollno, one for marks.

Is it possible to take different, different datatypes of all these variables under a single name →?

Yes, that is using a structure.

integer → is a datatype.

array → is a derived datatype

structure is also a datatype, but it is an User-defined datatype.

↳ means we can define our own datatype.

How can we define our own datatype →?

↳ using the primitive datatypes: int, char, float, double...

we can define our own datatype.

→ Let us say the following information we have about a student of different datatypes:

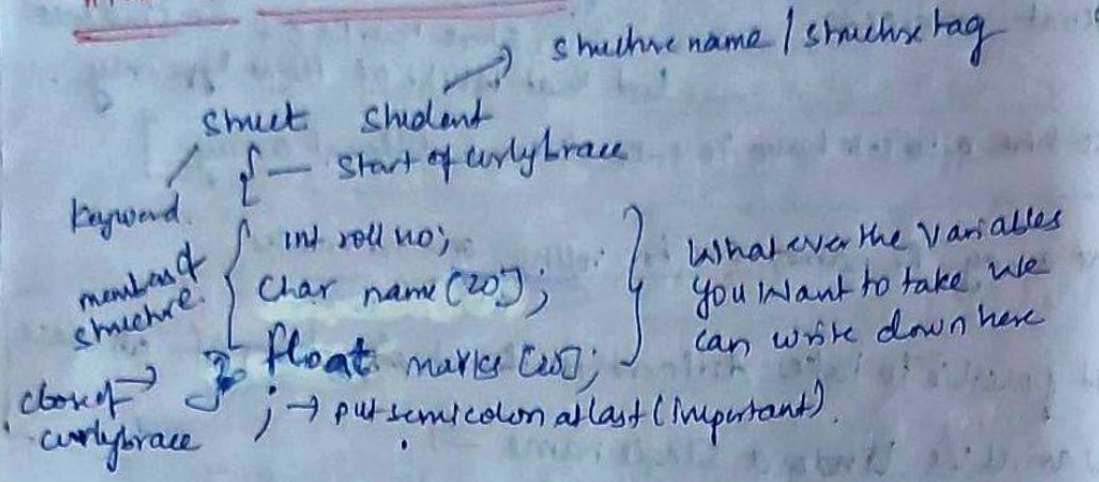
int rollno;
char name[20];
float marks;

} can we store the different different datatypes under a single name, under a single datatype →?

Yes that is a structure.

or we can say that is possible through a structure.

How to declare a Structure -)



This is the declaration of the structure.

Using the above, we can store the student information like for 60 students or for how many students you want to store you can store.

How you will store -) you have to define the objects of the structure.

array of objects also we can define.

Why we need structure -)

> If we need to process data of different, different datatypes we need structure.

In real life / real world if you want to develop a software or anything, you have to process data i.e. of different different datatypes.

→ So we cannot declare so many variables, in your program, that would be very lengthy, we cannot take even arrays, arrays is also not a good solution to this kind of data like students data, employees information.

In trees, Linked lists, if you want to process these kind of datastructures then also we need structure

Defn:

Structure is a user-defined datatype that groups elements or variables of different different datatypes under single name.

(or)

Structure is a collection of one or more variables of different different datatypes that are grouped together under a single name.

Important points about structures:

1. You have to put a semicolon at the end of structure ie. after closing curly brace.
2. In the members of structure, two members cannot have the same name. Ex: int rollno; float rollno; if you declare then it is wrong.
3. We cannot initialize individual structure members in the declaration

```
struct student
{
    int rollNO = 1;  → is not allowed, because
    char name[20];
    float marks;
};
```

if you declare the structure like this that doesn't mean the memory has been allocated to this structure - NO memory is not allocated, this is just a template, this will tell the Compiler how the structure is layed out / how the structure is there.

But memory is not allocated to the variables declared in structure.

`struct student` → this is just a datatype

Like suppose if I write int, float .... these are datatypes → does memory will be allocated to datatypes?

NO

if you write int a; → now memory will be allocated to this a i.e. 4 bytes.

So this `struct student` → is just a datatype, it is similar to writing "int. (so memory has not been allocated to this).

If memory has to be allocated to the structure?

you have to write down the datatype and then variable name followed by it

like int a;

`struct student s1;` → if you write like this memory will be allocated.

↳ s1 is called object of the structure

How much memory will be allocated?

Calculate the size of datatypes you mentioned in structure declaration (how many you declared)

for ex: in above rollno will take 4 bytes, char-20 bytes and float 4 So total 28 bytes are allocated for s1.

Declaring variables / objects of Structure.

→ structure is a user-defined datatype.
→ If it is a datatype then obviously, we can declare a variable of that data type (like int is a datatype and if you want to take a variable of integer the how we will declare →)

```
int a;
```
datatype ↓ variable

```
float b;
```
datatype ↓ variable

```
char c;
```

→ The above way we can declare variables of the above datatypes.
→ structure is also a datatype, but it is user defined, so obviously we can take the variables of the "structure datatype" but how you will declare those variables →)

Variables in structure are also called as objects in the structure.

Examples of structures: (valid / Invalid →)

①
```
struct student {
    int rollno;
    float marks = 60.5;
    char name[20];
    char address[30];
};
```

②
```
struct abc {
    int x;
    int y;
    int z;
};
```

③
```
struct xyz {
    int x;
    float y;
    char x[10];
};
```

④
```
struct Emp
{
    char *empname;
    int empid;
};
```

How to declare the variables of a structure: Method 1

Let's take an example:

```
struct student
{
    int rollno;
    float marks;
    char name[20];
};
```

struct is a keyword and student is structure tag. which is optional.

rollno, marks and name are members of the structure

→ Till now No memory has been allocated to the above structure...

because `struct student` → complete thing is just a datatype.

Suppose if i write in mains function. if you want to take a variable of type integer.

```
void main()
{
    int a;   // memory of 4 bytes will be allocated.
}
```

If i want to declare a variable of structure datatype → then you will write:

datatype followed by a variable name

`struct student` is the complete datatype here.

```
void main()
{
    int a;
    // struct s;  // students write simply struct and this is datatype, it is wrong,
    struct student s;   // and s - is variablename. Only this is correct
}
```

⇒ This is how you can declare a variables of a structure.

's' is also known as object of the structure.

Using this object only we can access the members of the structure.

At this time, now the memory has been allocated.

How much memory will be allocated -?

```
struct student
{
    int rollno;
    float marks;
    char name[20];
};
```

} calculate the memory required for these structure members.

```
void main()
{
    int a;  // 4 bytes of memory to a
    struct student s;
}
```

Now for s → how much memory will be allocated -?

the datatype of 's' is "struct student" and student is a user defined datatype, so how much memory will be allocated -?

you have to calculate the memory required for these structure members.

for rollno - 4 bytes, marks - 4 bytes, char 1 byte. and 20 × 1 = 20 bytes

So total 28 bytes will be allocated to s.

→ If you want to know the size of the structure,

```
printf("%d", sizeof(s));
printf("%d", sizeof(struct student));
```

Second method of declaring variables of a structure.

In the 1st method we saw we have declared the variable for a structure i.e. 's' in main() function

The second method is:

While declaring this structure only / while defining that structure datatype then only we can write down the name of the variable. i.e. 's' & no need to write down in the main() as shown below.

```
struct student
{
    int rollno;
    float marks;
    char name[20];
} s; // This s is the variable of datatype
                    "struct student". or
                    we can say 's' is the object of
                    the structure.
void main()
{ int a;
}
```

> s is the variable of datatype "struct student"

Note:
But generally the recommended method of declaring a variable of structure is 1st method.
i.e; in main() function.

```
struct student s;
```

We define variables in main() function, by defining the datatype of structure before main() function and its variable in main() function.

If you want to take 2 or 3 variables of structure type:?

```
struct student
{
  int rollno;
  float marks;
  char name[20];
} s1, s2;   // here only we can declare the variables here
            // if you want to take 2 or 3 variables of same type.
void main()
{
  int a;
}
```

Suppose you want to store information of 60 students-?

It is not like you will declare s1, s2, s3 ---. s60 till 60, this is not a good idea.

We will take an array of the structure variable.

Name of the structure / Structure tag is optional.

You can write like below also.

```
struct {
  int rollno;
  float marks;
  char name[20];
} s1, s2;
```

But the drawback of this way of defining the datatype of structure is what-?

At some point of time, i want to declare more variables, any where, it is not possible to declare the variables of this datatype in main() function. or any where in the program. because we don't have tag.

While declaring the structure variable in main() function take care as:

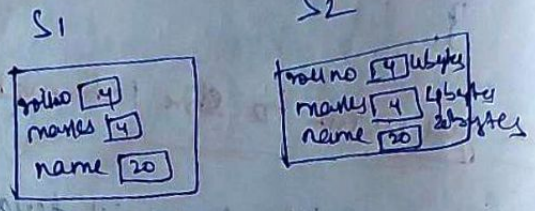
```
Struct Structuretag Variablename;
```

In the above declaration of structure, like did not mention structuretag / the name of the structure i.e. why

> It is not possible to declare a variable by the first method using the above way of declaring a structure.

Example:

```
Struct student{
    int rollno;
    float marks;
    char name[20];
};
Void main()
{
    int a;
    Struct student s1,s2;
}
```



→ can we declare like this?
yes.

Here for s1 object, a separate memory will be allocated, for s2 object a separate memory will be allocated

for s1 → 28 bytes of memory is allocated and for
s2 → 28 bytes of memory is allocated.

if one more variable then for 3rd also 28 bytes will be allocated.

Initialization and Accessing of a structure:

Main points → How to initialize the structure.
→ How to access the members of the structure.
→ How to assign values to the structure members.
→ How we are going to access those members and how we are going to print the values whatever you have assigned.
→ Compile time initialization, Run-time initialization.

How to Initalize a structure:

Let us discuss with an Example:

```
Struct student
{
    int rollno;
    float marks;
    char name[20];
};
Void main()
{
//Struct student s;
```

// how to initialize the above members of the structure?
// While declaring the variable of the structure here only we can initialize i.e. Compile time initialization.

```
Struct student s = { 1, 50, "bhargav" };
```

↓ We have to initialize how the way we have declared the members of structure in defining a structure.

This is what is Compiletime initialization of the structure

There are some rules of Compiletime initialization.

Rules of Compile time Initialization:

1. ordering of variable initialization should be the same order how we declared the members in a structure definition. It should match.
2. If you do partial initialization i.e if you initialize one value i.e to one structure member and don't initialize the other values to default it i.e. {null, 0}.

By default, int and float will be assigned to 0' and string and character will be assigned with null.

void names { struct student s={ };

3. If you don't want to declare the variable in the main function and want to declare it at the end of structure declaration, then only we can initialize the structure.

```
struct student
{
    int rollno;
    char name[20];
    float marks;
} s={1, "bhargav", 90.9};
```

// suppose you want to take two variables, two objects s1 & s2,

```
struct student
{
    int rollno;
    char name[20];
    float marks;
} s1={1, "bhargav", 90.9};
  s2={2, "bharat", 92.1};
```

This is wrong

We can't Initialize like this

after this semicolon, s2 will not be considered as object of this structure / the variable of the structure

How to initialize two variables / more of the same structure:

```
struct student
{ int rollno;
  char name[20];
  float marks;
};
Void main()
{
   struct student s1={1, "Lakshman", "95"};
   struct student s2={2, "bhargav", "98"};
   /* How many variables/objects you want, you
                  can initialize like this */.
}
```

But if you want to store 60 variables or let us say i want to store information of 60 students, so then it is not a good idea to take s1, s2 ---- s60 students.

In this case we have to take array of 60 variables/objects.

We can initialize like below also:

```
struct student
{
  int rollno;
  char name[20];
  float marks;
} s={35, "bharat", 70};
Void main()
{
  struct student s1={1, "shiva", 90.91};
  struct student s2={2, "bhargav", 95};
}
```

We can initialize outside of the mains functions and inside main() function too.

We can initialize like below too:

```
struct student
{
  int rollno;
  char name[20];
  float marks;
} s = {1, "Lakshmi", 90};

struct student s1 = {2, "bhargav", 95};
struct student s1 = {3, "bharat", 97};
void main()
{
  ----
}
```

It is not compulsory that you are supposed to write down the statements of initialization in main() function only outside of the main() also we can declare, initialize the variables of a structure.

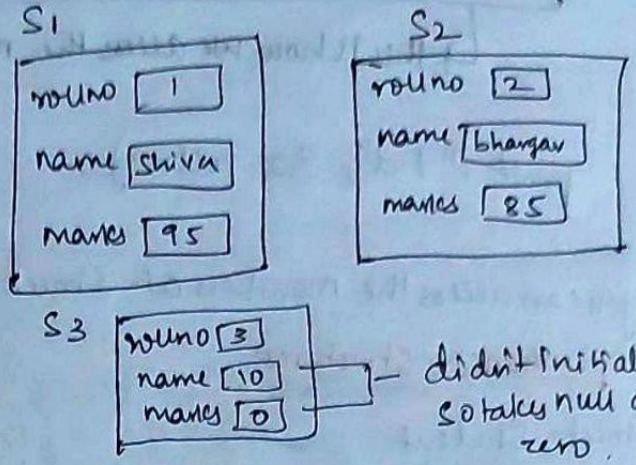
Note:

If you declare the structure outside of the main() it is global, you can access this structure in any function in main() or any other functions defined in the program. and in any function you can declare these type of variables. and you can access the members of the structure.

How to access the members of a structure:

Suppose i want to print the values of S1 and S2.

```
struct student
{
    int rollno;
    char name[20];
    float marks;
};
```




```
void main()
{
    struct student S1 = {1, "Shiva", 95};
    struct student S2 = {2, "bhargav", 85};
    printf("%d", rollno);
```

→ this is wrong.

→ individual rollno we cannot write and it will print rollno value here – NO (Which rollNO it will print?)

```
struct student s3={3};
```

Imp: Simple rollNO doesn't have any meaning here:

You cannot access the members of a structure by writing the name only.

These are not simple variables.

→ you are supposed to link the member variables of the structure with the structure object / structure variable.

Imp: `printf("%d", S1.rollno);` // now this is having some meaning, it is what → rollno of S1, it is a variable representing rollno of S1. and this is how we can access the members of structure.

Object name dot operator (member operator) membername.

↳ This is how we access the members of structure.

printf("%d", s2.rollno);

You can access the members of structure using "dot (.)" operator

Ex: operations on structures:

```
Struct Student
{
  int rollno;
  char name[20];
  float marks;
};
void main()
{
  Struct Student s1; // i am not initializing here
  Struct Student s2 = {2, "Shiva", 85};
  s1 = s2; // this will work, you can copy these values.
```

↳ So here whatever the values are there in s2, these values will be copied into s1 also.

So in s1 also we will have 2, "Shiva" and 85.

s1 = s2; → is valid because they are of same type (data type) Struct Student.

but if you have two structures like Struct employee and Struct Student for that we declare a variable e and s students then e = s1 ✗ wrong. → it not same type not allowed.

Compile time Initialization of structure.

Ex:

```
struct student
{ int rollno;
  char name[20];
  float marks;
};
void main()
{
  struct student s1;
  struct student s2={ 2,"shiva", 85};
    s1.rollno=1;
    s1.name="bharat";
    s1.marks = 90;
}
```

So individual members also we can initialize but like this.

The above is compile time initialization

Run-time Initialization of structure:

```
struct student
{ int rollno;
  char name[20];
  float marks
};
void main()
{ struct student s1;
  struct student s2={ 2, "jiya", 85};
  printf("Enter information for s1");
  Scanf("%d", &s1.rollno);
  Scanf("%s", s1.name);
  Scanf("%d", &s1.marks);
}
```

This is what is run-time initialization.

Operations on Structure:

Example:

```
Struct student
 {
   int rollno;
   char name[20];
   float marks;
 };
Void main()
 { struct Student s1={1,"Shiva", 90};
   Struct Student s2={2, "bhargav", 95};
// if (s1 > s2)  } ⇒ you cannot compare like this.
or if(s1<s2)  }
   if(s1==s2).  } ⇒ this is wrong.
}  if(s1!=s2)  }
```

> But you can compare individual members of these Objects.

How we can compare individual members of these objects?

Example:

```
Struct student
  {
    int rollno;
    char name[20];
    float marks;
  }
Void main()
  {
```

```
struct student s1 = {1, "shiva", 90};
struct student s2 = {2, "bhargav", 95};
   if (s1.rollno < s2.rollno)
   {
      printf ("Hello");
   }
}
```

Like this we can Compare.

Individual members of the structure we can Compare

rollno we can compare, marks you can compare and strings you can compare using string comparison functions,

But directly we cannot compare the objects / variables of structure.

Program for compile time initialization

```
#include <stdio.h>
  struct student
  { int rollno;
    char name[20];
  }; float marks;
Void main()
{
struct student s1 = {1, "sai", 90};
struct student s2 = {2, "Jiya", 95};
printf("info for s1");
printf("\n%d %s %f", s1.rollno,
      s1.name, s1.marks);
printf("info for s2");
printf("\n%d %s %f", s2.rollno,
  s2.name, s2.marks);
}
```

o/p: 1 sai 90
2 Jiya 95

```
#include <stdio.h>
  struct student
  { int rollno;
    char name[20];
    float marks;
  };
Void main()
{ struct student s1 = {1, "Jiya", 95};
  struct student s2;
  s2 = s1;
printf("info for s1");
printf("\n%d %s %f", s1.rollno,
      s1.name, s1.marks);
printf("info for s2");
printf("\n%d %s %f", s2.rollno,
  s1.name, s1.marks);
}
```

o/p: 1 Jiya 95
1 Jiya 95

Program:

```
#include<stdio.h>
struct student
{ int rollno;
  char name[20];
  float marks;
}s3={2};
void main()
{
  struct student s1={1,"lakshmi",95};
  struct student s2;
  s2=s1;
  printf("info for s1");
  printf("\n %d %s %f",s1.rollno,s1.name,s1.marks);
  printf("\n info for s2");
  printf("\n %d %s %f",s2.rollno,s2.name,s2.marks);
  printf("\n info for s3");
  printf("\n %d %s %f",s3.rollno,s3.name,s3.marks);
}
```

Op:

s1 & s2 will print the same values.

s3 will print 2 0.000000

Operations on struct variables in C:

In, C the only operation that can be applied to struct variables is "assignment"

Any other operation (ex: equality check) is not allowed on struct variables.

Program 1:

```
#include <stdio.h>
 struct sample
{
 int x;
 int y;
};
int main()
 {
   struct sample s1={10,20};
   struct sample s2 = s1; // contents of s1 are copied to s2
   printf("p2.x=%d, p2.y=%d", p2.x, p2.y);
   getchar();
    return 0;
  }
```

Program 1 works without any error.

Program 2:

```
#include<stdio.h>
  struct sample {
      int x;
    }; int y;
  int main()
    { struct sample s1={10,20};
struct sample s2 =s1;
if(p1==p2) // Compiler error: Cannot
// do equality check for whole structures
 {  printf("p1 and p2 are same");
 }
  getchar();
 return 0;
}
```

Program 2 fails in compilation

Operations on Individual members of structure:

All operations are valid on individual members of structure.

Program illustrating Operations on structure members:

```
#include <stdio.h>
#include <string.h>
struct student
  { int rno;
    char name[10];
    int marks, age;
   };
  void main()
   { int n;
// assigning values to structure variable s1 using initialization.
struct student s1 = { 2, "Shiva", 89, 18};
 struct student s2;
  s2 = s1;
printf("\n Details of student 1: \n");
 printf("\n roll number: %d", s1.rno);
 printf("\n name: %s", s1.name);
  printf("\n marks: %d", s1.marks);
 printf("\n age : %d", s1.age);
  printf("\n\n");
printf(" details of student 2: \n");
printf("\n roll number: %d", s2.rno);
printf("\n  name: %s", s2.name);
```

```
printf("\n marks: %d", s2.marks);
printf("\n age: %d", s2.age);
// comparision of two student details.
m = ((s1.rno == s2.rno) && (s1.marks == s2.marks))?1:0;
if (m==1)
    printf("\n both the details are same");
else
    printf("\n both the details are not same");
}
```

output:

Details of student 1:
roll number: 2
name: bhargav
marks: 89.
age: 18

Details of student 2:
roll no: 2
name: bhargav
marks: 90
age: 18.

both the details are same

⇒ Nested structures in C:

In this we declare a structure variable as a member of another structure.

We will consider the following example to illustrate this concept:

I want to create/define a structure of a student.

let us consider student details. In date of birth we can have.

we will have attributes: student id, sname and date of birth.

again: day, month, year } on which they are born.

This is the structure we need to create.

We have to create a student structure:
in which student contains student name, student rollno
and dateofbirth.
and dateofbirth should again contain day, month, year.

So for the above, we need to define two structures and we have to declare one structure variable as a member of another structure

so first we have to create a structure with id, name & dateofbi
and his dateofbirth is again defined as a structure with the
members: day, month and year.
We create dateofbirth structure variable as a member of
structure.

First we create dateofbirth structure and then studentstructure,
because we are declaring "dateofbirth" structure variable as a
member of structure student.

```
// creating a structure for dateofbirth.
struct dob
{
    int day;
    int month  // if you want to use characters you have
               // to write an character array like
    int year;  // char month[20];
};  // here we have created a structure but not a structure
    variable.
// Now we will create the structure for the student
as below.
```

```
struct student
{
    int id;
    char name[20];
    struct dob d1;
};  // not declared any struct variable for student
main()
{
    struct student s1;
// s1 will be having
// id name and date of birth. and this. dob have day,
// month and year
// so inorder to access the member of structure, we need
// a structure variable and the dot operator.
// so inorder to access the id is "s1.id", inorder to access the
// name "s1.name" ; in order to access d1 if it is s1.d1 and again
// d1 is a structure variable, so it has to follow the structure of
// date of birth and in this we have 3 members, you have to
// access these 3 members, again we have to use the dot operator
// and structure variable, i.e.
    d1.date; d1.month; d1.year.
```

// Now dob is again a structure, now as a member of a student, we have to declare the dob structure variable. So inorder to create a structure variable we need to take the reference of tag name this d1 we have declared as a member of another structure.

Again In order to access the student with d1 we have to access like!

```
s1.d1.day;
s1.d1.month;
s1.d1.year;
```

In order to access the member of a structure, we require the structure variable and the dot operator.
only by means of the dot operator only we can access the members of structure

so here also we are accessing the members of structure using s1.d1.day / s1.d1.month / s1.d1.year

struct variable ! dot operator

```
// Rest of the program.
    printf("Enter id:");
    scanf("%d", &s1.id);
    printf("Enter name");
    scanf("%s", s1.name);
    // In order to access date of birth.
    printf("Enter date of birth");
    scanf("%d %d %d", &s1.dt.day, &s1.dt.month, &s1.dt.year);
    printf("id=%d", s1.id);
    printf("name=%s", s1.name);
    printf("dob=%d-%d-%d", s1.dt.day, s1.dt.month,
                           s1.dt.year);
}
```

This is the simple program to illustrate nested structure.

// one more way of Nested Structure:

In a structure we can directly use the structure in place of "a structure variable" i.e. "implementing a structure within a structure".

```
struct student
{
    int id;
    char name[20];
    // Instead of creating a structure variable we can define
    a structure here itself
```

```
struct dob
{ int day;
  int month;
  int year;
}dI;
};
```

This is the alternate way of implementing a nested structure.

We can use any one among the above two ways.

→ Both are correct.

→ We can define the structures separately and we define the structure variable inside the structure.

→ We can define one structure and inside this structure we can define another structure.

complex structures in C:

"When you'r structure will be called as complex structure?

" Basically if you have a nested structure i.e. a structure within a structure "

→ If your one structure contains another structure, this is called as nested structure and it is one of the form of complex structure.

→ Another possibility is that, if your structure contains an array i.e array within a structure, this is also a form of nested structure (a complex structure).

Another Example of nested structure:

→ Considering the details of employee, we want to store his/her joining date and date itself can be a structure means, He can represent a date in the form of a structure,

→ They are: day can be 1 element, month can be 1 element, and year can be 1 element.

→ The date structure we can include it in employee structure

Example:

```
struct date          // this date structure variable we will use
{                       inside the employee structure i.e. why
                        we have to define this structure
  int day;              first.
                        we have to follow the order.
  int month;
  int year;
};

struct employee
{
  int id;
  char name[20];
  float salary;
  struct date join_date;
};

void main()
{
    struct employee e1;
    ;  similar to student nested structure
}                       ex: assignment to students.
```

// Another way of defining nested structure.

```
struct employee
{ int id;
  char name[20];
  float salary;
  struct date
  { int day, month, year;
  } joindate;
};
```

What is the disadvantage of this?
This is a kind of local structure
Scope of this structure is
applicable for employee only.
But if you declare this outside
like before it is available
to your entire program.

Suppose you define another structure manager and in that you use joining date, in this case also we can use the same "date" structure here also we not define it again.

Accessing the elements of nested structure:

```
main()
{ struct employee e;
  ....
  printf("Enter id, name, salary");
  scanf("%d %d %d", e.id, &e.name, &e.salary);
  printf("enter joining date");
  scanf("%d %d %d", &e1.joindate.day, &e1.joindate.
        month, &e1.joindate.year);
  printf("id=%d, name=%s, dob=%d-%d-%d",
         e.id, e.name, e.joindate.day, e.joindate.year,
         e.joindate.month); }
```

Array within structure:

You can say we have already declared an 1d array in nested structure i.e. a single character array.

But a single character array is treated as a string.

→ Here there is no complexity.

→ But if that array is of type "numeric value" then → complexities get increased.

→ That's why it is treated as / comes under the category of complex structure.

→ Assume that i have a student structure, and i want to store the student details.

Ex:

```
struct student
{
    int rollNO;
    char name[20];
    char course[10];
    // i want to store marks of 5 subjects, so instead of
storing [defining marks1, marks2, marks3, 4, 5, what we
will do?
    int marks[5]; → instead of declaring 5 different
};
variables, i will say marks[5], this particular form
is nothing but array within structure, Here we
use arrays numerically, not a character array.
```

Here two character arrays we declared are treated as single strings → these do not add complexity in your structure, But whenever you are adding numeric array, definitely it is going to add complexity in your structure i.e. why it is an example of complex structure i.e. Array within structure.

// Now how to access these elements.

Suppose we want to input the marks of students.

So I will create a student variable: s

```
main()
{
  struct student s;
  printf("Enter roll no");
  scanf("%d", &s.rollno);
  printf("Enter name & course");
  scanf("%s %s", s.name, s.course);
  // printf("Enter the marks of 5 subjects");
  // it is an array so we have to use loop.
  for(i=0; i<5; i++)
  {
    scanf("%d", &s.marks[i]);  // it will scan the
  }                            marks of one student s for 5 subjects.
```

Similarly for another students say s1, s2 --- we need to scan marks then what we need to do -?

we need to write a loop and then s1.marks[i], s2.marks[i] — this is complex.

So this is an example of a complex structure.

```
// printing the marks.

for(i=0; i<5; i++)
{
    printf("%d", s.marks[i]);
}
}.
```

Another Example illustrating array within a structure:

- A structure is a datatype in C (user defined), that allows a group of related variables to be treated as a single unit instead of separate entities.
- A structure may contain elements of different datatypes - int, char, float, double etc.
  - → It may also contain an array as its member
- → such an array is called array within a structure.
- → An array within a structure is a member of the structure, and can be accessed just as the access just as we access other elements of the structure.

The program demonstrating the array within a structure.

→ program displays the record of a student comprising the rollno, grade and marks secured in various subjects.

→ The marks in various subjects have been stored under an array called marks.

→ The whole record is stored under a structure called a

Program:

```
#include <stdio.h>
struct candidate {
    int rollno;
    char grade;
    // Array within the structure
    float marks[4];
};
void display(struct candidate a1)
{
    printf("Rollno : %d\n", a1.rollno);
    printf("Grade : %c\n", a1.grade);
    printf("marks secured :\n");
    int i;
    int len = sizeof(a1.marks) / sizeof(float);
// Accessing the contents of the array within the structure.
```

```
for(i=0; i<len; i++)
  {
    printf("subject %d: %0.2f\n",
            i+1, a1.marks[i]);
  }
}

int main()
{
    // Initialize a structure.
struct Candidate A = {1, 'A', {98.5, 77, 89, 78.5}};

    display(A);
    return 0;
}
```

Op: Roll no: 1
Grade: A
marks secured:
Subject 1: 98.50
subject 2: 77.00
Subject 3: 89.00
subject 4: 78.50

Array of structures:

- What is an array of structures
- Why we need an array of structures.
- How memory will be allocated to those members of structure.
- How you can access those structure members.

> If in your program you are using array of structures, if you know how memory is allocated and how to access those structure members. then it is very easy to write a program.

What is an array of structures?

Let us take an example:

```
struct student
{
    int rollno;
    char name[20];
    float marks;
};
```

These are structure members and their rollno, name, marks are not variables we can't say them as variables, within structures these are not simple variables, these are structure members.

```
struct student s;
```

(struct student = datatype; s = this is what is a variable (object).)

Suppose if I want to store information of two students you can simply declare two variables like this s1, s2. if 3 then s1, s2, s3... so on.

```
struct student s1, s2;
```

and you can individually enter the information of 3 students and you can just print.

But, the problem comes when we store information of 60 students.

We have to declare 60 variables, but it is not a good idea. How many variables we have to declare? 60 and it would be a very lengthy program. because of the declaration of the variables only and obviously it will be confusing to remember the name of all the variables.

So what we can do?

We can take an array of these variables (of 60 ... soon).

i.e. exactly known as "array of structure"

So we can write as:

```
struct student
{
    int rollno;
    char name[20];
    float marks;
};
struct student s[60]; // this is what? an array of 60 variables.
```

It is an array of 60 variables and each array element is representing the structure object. Indirectly we can say we are having 60 structure objects / 60 variables of this structure.

So why we need array of structures is clear, i.e. if you want to store more information of the student, more than 5 / 10 students i.e. larger no of students / large no of employees of a company there we need array of structures.

Why we need structure-) Because the information is having different different datatypes

Why we need array of structures-)

Because the quantity of the information stored (i.e no of students info, or the no of employees information) i.e very large. i.e. why we need array.

We will take 3 students information:

```
struct student;
{ int rollno;
  char name[20];
  float marks;
};
struct student s[3]; // you can also declare this in main() function.
```

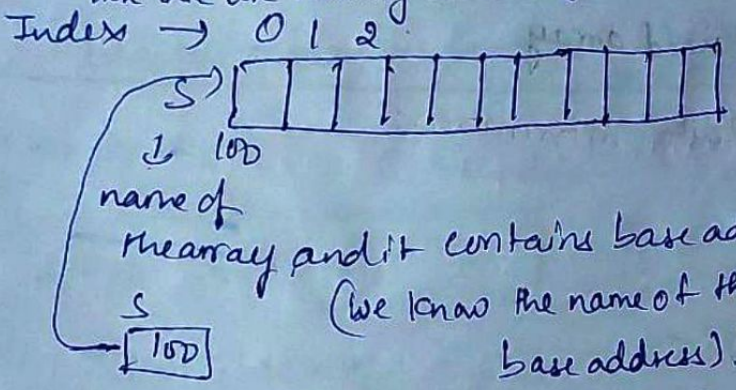
How the memory will be allocated → It is very important.

How to access the structure members and how to initialize these structure members is also very important.

P.T.O

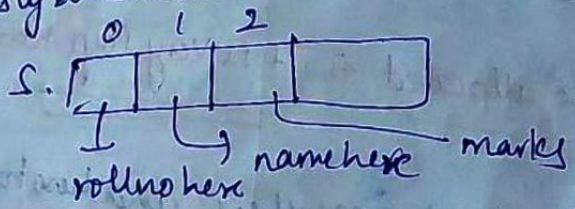
```
struct student
{
    int rollno;
    char name[20];
    float marks;
};
void main()
{
    struct student s[3];
```

Here we are having an array.



How you will store rollno, name and marks information?

Obviously we cannot store like

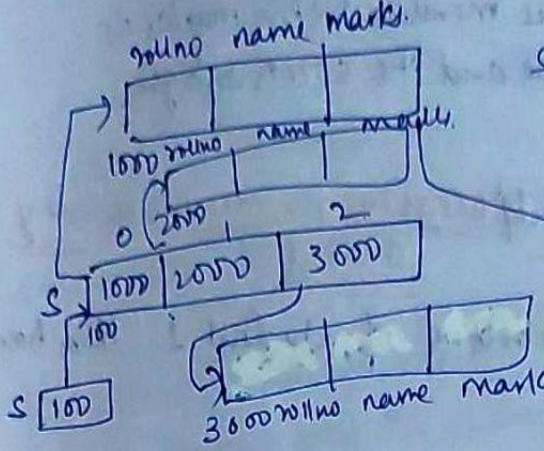


How you will store the information?

How you can access?

* → So this each array element is a structure object and each structure object is having information of structure members i.e. rollno, name and marks. also

Let us illustrate this



Suppose base address of rollno is 1000

this s[0] will point to 1000. → here we can store the information for the first object i.e. s[0]. i.e. first student.

→ for second student, obviously we need to store 3 information, i.e. rollno, name & marks.

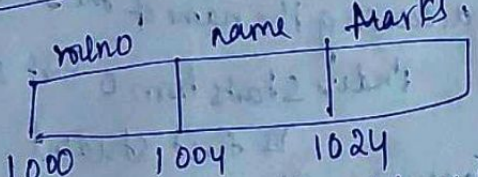
→ so in a separate memory locations, memory will be allocated for rollno, name and marks.

Suppose base address is 2000, so s[1] will be containing the address 2000

→ for third student, suppose the base address is 3000 so s[2] will contain the address 3000

So this is how this will be stored in memory.

Now ⇒



In this

↳ Suppose base address is 1000 now next address will be 4 bytes i.e. 1004 because the type of rollno is int and next element (it is name) its base address will be "1004" because the type of previous member is int in structure defined.

(rollno) (type)

The address of marks will be 20 bytes after sc2 i.e. 1024. (previous member before marks is name and its size is 20 bytes.

Accessing the values of student s/ Accessing the members of every student.

→ If we want to access the rollno of student 1, then how we can access →)

Obviously using

> structure variable name, the dot operator and then the structure member name →)

↳ this is for simple variable name

But here we are not having simple variable name.

> We are having array.

So how to access →)

```
s[0].rollno;   // accessing the rollno of 1st student
s[0].name;     // accessing the name of the 1st student
               // index starts from 0 for 1st student
               // 1st student.
s[0].marks     // accessing student1 marks.
```

If you want to access the information of student 2,
→ Base address is 2000 and is stored in s[1].

So we access like:

```
s[1].rollno;
s[1].name;
s[1].marks;
```

Similarly for student 3: s[2].rollno, s[2].name, s[2].marks.

So this is how Members of student structure is stored in memory for array of structures and the above way we access the information about students.

Complete program illustrating array of structures:

We are entering information about students and we are going to print that information.

```
struct student
{ int rollno;
  char name[20];
  float marks;
};
void main()
{ struct student s[3]; int i;
// To enter the information in array we have to use loops (storing
// 3 students details
for(i=0;i<3;i++)
{ scanf("%d %s %f",&s[i].rollno, s[i].name,
        &s[i].marks);
}
```

// string so we will not give &.

(or)

// If we want to enter the information about individual students then you have to give printf statement stating that enter information about student 1 & student 2, student 3 and we have to scan individually.

// printing the information about 3 students.

```
for(i=0; i<3; i++)
{
    printf("\n rollno = %d  name = %s  marks = %f",
            s[i].rollno, s[i].name, s[i].marks);
}
}
```

This is how we can use array of structures.

Array of structures program using Static Initialization

→ In the following demonstration, the array holds the details of the students in class which include, the rollno, grade, which have been grouped under a structure(record).

– There exists one record for each student.

– This is how a collection of related variables can be assembled under a single entity for enhancing the clarity of code and increasing its efficiency.

program:

```
#include<stdio.h>
struct class {
    int roll_no;
    char grade;
    float marks;
};
```

```
void display(struct class student_record[3])
{
    int i, len=3;
    for(i=0; i<len; i++)
    {
        printf("RollNo : %d\n", student_record[i].
        rollno);
        printf("name : %c\n", student_record[i].grade);
    printf("Average marks : %0.2f\n", class_record[i].
                                        marks);
        printf("\n");
}
main()
{
    // Initialization of an array of structures
    struct class class_record[3]
    = {{1, 'A', 89.5f}, {2, 'C', 67.5f},
    {3, 'B', 70.5f}};
    // function call to display the
    display(class_record);
    return 0;
}
```

Op:

Roll number : 1 Grade : A Average marks : 89.50

Roll number : 2 Grade : C Average marks : 67.50

Roll number : 3 Grade : B Average marks : 70.50

Difference between the "Array within a structure" and Array of structures:

S.no	Content on which both can be differentiated.	Array within a structure	Array of structures
1.	Basic idea	A structure contains an array as its member variable.	An array in which each element is of type "structure"
2.	Access	Can be accessed using the dot operator just as we access other elements of the structure.	Can be accessed by indexing just as we access an array.
3.	Syntax	struct class { int ar[10]; } a1, a2, a3;	struct class { int a, b, c; } students[10];

pointers to structures:

We will understand the concept of pointers to structure contents.

→ How to access the members of a structure using pointers.

→ We have to create a structure first.

→ We have to declare a structure variable as well as a pointer variable. and then we have to initialize the pointer variable with a structure ad

by using these pointers we have to access the members of the structure?

Recap of pointers Concept:

We know in pointers concept, we use two operators i.e. '*' and '&' ⇒ when '*' it's indirection operator and '&' ⇒ Address of operator.

→ Here we have to declare a pointer and initialize the pointer with some variable, that means pointer variable is a variable which holds the address of another variable.

→ So first we are declaring a variable and then we are declaring a pointer variable. and we will initialize this pointer variable with address of another variable.

```
int a; int *p; p=&a;
```

The same procedure will be followed in structures also

1. So first we have to define the structure
2. We have to declare a structure variable and declare a pointer variable
3. Assign the address of the structure variable with a pointer variable
4. We have to access the members of a structure using this pointers

Steps: ① Define structure
② Declare structure variable
③ Declare pointer variable
④ Initializing pointer variable
⑤ Access the members of structure through pointer.

example:

first step → defined the structure

```
struct student
{
    int rollno;
    char name[20];
    float percentage;
} s1; // created one structure variable s1.
```

Note: We can create a structure variable at the definition itself or inside the main function, anywhere we can create the structure variable.

⇒ Now pointer variable has to be declared, so how to declare a pointer variable?

⇒ In a normal (integer pointer) or (pointer) every pointer should have a datatype.

↳ whatever the address we are storing into the pointer variable that datatype should be used for the declaration of this pointer.

i.e. if a pointer is used to hold the address of integer variable, we have to declare that pointer with an integer datatype.

→ If a pointer is used to hold the address of floating point variable, we have to declare the pointer with a floating point datatype.

→ similarly here, we are supposed to store the address of a structure variable.

So the pointer should be declared using a structure.

Inorder to declare a structure variable, the syntax is:

```
struct structure tag  structure variable / variablename
```

So Inorder to declare a pointer which stores the address of a structure, the syntax is:

```
struct structure tag *ptr;
```

⇒ here "ptr" is a pointer variable which is declared using a structure and it stores the address of a structure variable which follows the student structure.

→ After the declaration, we have to initialize the pointer variable with address of the structure variable, to the pointer variable.

```
ptr = &structurevariable;
```

```
struct student
{
    int rollno;
    char name[20];
    float percentage;
};                                  Defining the structure.
struct student s1;          // Declaration of structure variable.
struct student *ptr;        // Declaration of pointer variable.
ptr = &s1;                  // Initialization of pointer variable
                            // assigning address of structure to pointer
```

// Accessing the members of Structure through Structure variable:

```
s1.rollno
s1.name
s1.percentage
```

This is the normal way to access the members of structure using structure variable.

But we have to access the members of a structure using the pointer variables but how?

Accessing the members of a structure through pointer variables.

Here we use an another operator called arrow operator →

In order to access the members of structure using the structure variables we use the dot (.) operator.

In order to access the members of a structure using the pointer variable we use the arrow operator.

In the keyboard press "minus" and "greater than" symbol to use arrow operator in the program.

So in order to access the rollno of student1 using pointer we write as:

```
ptr → rollno;
ptr → name;
ptr → percentage;
```

} this will give the values of members of structures using pointers.

both accessing the members of a structure using the structure variables and accessing the members of a structure using pointers will give the same result.

but dot operator is used in normal access (i.e. structure variable) and arrow operator is used in pointer access.

program:

We will initialize only one variable and then let us initialize the pointer variable and access the structure members using the pointer

```
struct student
{
int rollno;
char name[20];
float percentage;
};
Void main()
{
  struct student S1;
  struct student *ptr;
        (or)
// we can declare in a single line too
    struct student S1, *ptr;
// Initialization of pointer variable.
  ptr = &S1;  // the address of S1 is initialized to
              // the pointer variable.
printf("Enter rollno, name, percentage");
scanf("%d %s %f", &S1.rollno, S1.name, &S1.percentage);
scanf("%d %s %f", &ptr->rollno, ptr->name, &ptr->percentage);
// printing the values using pointers / Accessing the values using
  // pointers
printf("Roll NO=%d", ptr->rollno);
printf("Name = %s", ptr->name);
printf("percentage=%f", ptr->percentage);
}
```

This is how we access structures using pointers.

Structures containing pointers / pointers as member of structure in C.

In our program if in a structure if we have any member as pointer then how can we access that pointer?

We will illustrate an example in which we define a structure in which we declare a pointer variable as a member and we will access that pointer member.

Program:

```
#include <stdio.h>
struct Info          // we have created a structure called Info and p is an integer pointer.
{ int *p;            // declared an pointer which stores the address of another variable whose datatype should be integer.
};
// we will create a structure variable and through this structure variable, we will access the pointer 'p'
```

If we want to create a pointer for structure then we can access the elements using arrow operator.

But here we are creating a structure variable and accessing p(pointer) using this variable, so we use dot operator.

```
struct Info Myvar; // created a structure variable.
// now i will try to access the pointer p using this structure variable Myvar.
```

our aim is to copy a value in a pointer 'p'.

and in a pointer if we have to store a value, then we give as

```
*p = someValue;
```

so we write as.

```
*(myvar.p) = 10;  // Here we have stored a value for the
                  // pointer variable using a structure
                     variable.
// we will print the value of pointer.
printf ("%d", *(myvar.p));
}
```

// In the program we have created a structure variable, and through this variable we have copied some value in pointer p and after copying i am printing the pointer value. to see whether the value we have copied is really copied or not.

If we execute the above program it will not give errors, but 1 warning will be given.

But when you execute this program, it gives you "segmentation fault" so,

We have to understand one very important point.

> In the structure declaration, if you have any pointer variable declared, in order to access this pointer before accessing this pointer, we have to allocate the memory to this pointer.

In order to allocate memory to pointer in a structure, we have two techniques.

1. allocate address of any other variable to this pointer.
2. allocate the memory at runtime using memory management function (i.e. malloc / calloc.

Let us solve the problem of the above program:

```
#include<stdio.h>
struct Info
{ int *p;
};
int main(void)
{
struct Info MyVar;
int d = 0; // we have declared a local variable.
// we will assign the address of this variable to p like below
MyVar.p = &d; // We have assigned the address of d to p. (Initialized).
*(MyVar.p) = 10;
printf("%d", *(MyVar.p));
return 0;
}
```

Op: 10

Note:

If you have any pointer variable as a member of structure, then if you want to access that pointer first of all you have to allocate a valid memory to that pointer (Initialize the address of another variable to this pointer).

// We can allocate the memory using heap too
We will remove int d=0; in the program and if you want to allocate the memory dynamically i.e. at run time, we have to call memory management functio

// modifying the above program by allocating the
// memory to the pointer variable at runtime using memory
management functions.

```
#include <stdio.h>
#include <stdlib.h>
struct Info
  {
    int *p;
  };
int main(void)
  {
    struct Info myvar; // created a structure variable
// allocating the memory to p using malloc
myvar.p = malloc(sizeof(int)); // as p is an integer pointer
// we have taken sizeof integer to allocate the memory to pointer
// and as it is malloc and its declaration is there in stdlib.h
// we have to include "stdlib.h" headerfile.

*(myvar.p)=10;
printf("%d", *(myvar.p)); free(myvar.p);
  return 0;
  }
```

Olp: 10

Note :- 1. When ever we are allocating the memory using memory management functions like malloc, calloc you as user has to free that memory

2. → Sometimes when malloc fails to allocate memory, then it will return "null", so we have to check whether proper memory is allocated or not so in order to check this we have to modify the above program as below.

```
#include <stdio.h>
#include <stdlib.h>
struct Info
{
  int *p;
};
int main(void)
{
  struct Info Myvar;
  Myvar.p = malloc(sizeof(int));
  if(Myvar.p == NULL)
    return -1;
  *(Myvar.p) = 10;
  printf("%d", *(Myvar.p));
  free(Myvar.p);
  return 0;
}
```

Op: 10.

// Accessing the structure member which is a pointer through a structure pointer.

Program:

```
struct test
{ char name[20];
  int *p;
};
struct test t1, *ptr = &t1;
```

// Here p is a pointer to int and a member of structure test.

There are two ways in which we can access the value (i.e. (address) of P:

1. using structure variable as: t1.P
2. using pointer variable as: ptr -> P.

Similarly, there are two ways in which we can access the value pointed to by 'P'.

1. Using structure variable as: *t1.P
2. Using pointer variable as: *ptr -> P

VIMP
Note:

1. The precedence of dot(.) operator is greater than that of indirection (*) operator, so in the expression *t1.P the dot(.) is applied before the indirection operator.
2. Similarly in the expression *ptr -> P, the arrow(->) operator is applied followed by indirection(*) operator.

Complete program illustrating the above concept:

```
#include <stdio.h>
struct student
{ char *name;
  int age;
  char *program;
  char *subjects[5];
};
```

```
int main()
{
    struct student stu = { "Aditya", 25, "CS",
                           { "CS-01", "CS-02", "CS-03", "CS-04",
                             "CS-05" }
                         };
    struct student *p = &stu;
    int i;
    printf("Accessing members using structure variable:\n\n");
    printf("Name: %s\n", stu.name);
    printf("Age: %d\n", stu.age);
    printf("program enrolled: %s\n", stu.program);
    for(i=0; i<5; i++)
    {
        printf("subject: %s\n", stu.subjects[i]);
    }
printf("Accessing members using pointer variable:\n\n");
    printf("Name: %s\n", p->name);
    printf("Age: %d\n", p->age);
    printf("program enrolled: %s\n", p->program);
    for(i=0; i<5; i++)
    {
        printf("subject: %s\n", p->subjects[i]);
    }
    return 0;
}
```

Olp:

Accessing membus using structre variables.

Name: Aditya

Age: 25

program enrolled: CS

Subject: CS-01

Subject: CS-02

Subject: CS-03

Subject: CS-04

Subject: CS-05

Accessing membus using pinter variable.

Name: Aditya

Age: 25

program enrolled: CS

Subject: CS-01

Subject: CS-02

Subject: CS-03

Subject: CS-04

Subject: CS-05

Structures and functions in C:

How structure variables can be passed as Parameters to the functions?

This can be done in 3 ways:

Call by value
1. Passing individual members of a structure.
2. Passing the entire structure.

Call by reference
3. Passing the address of a structure.
↳ here we pass address of member variable instead of value
↳ here structure variable

So there are three ways we pass the structures as arguments to the functions (User defined functions).

Passing individual members of a structure:

Rather than passing the entire structure, we will pass the individual members of the structure here.

In functions we will have:
1. function declaration
2. function calling
3. function definition.

Inside the function call we have to write the arguments / we have to specify these arguments.

function declaration will be outside the main function
function call will be inside the main() function
function definition is outside the main() function
i.e. where the logic you have to write.

So, in the function call, we are specifying the arguments, so it makes a reference, whenever the control of program reaches this function call, automatically the control moves to the function definition & it gets executed and immediately after that control goes back to to parent position

Same logic is applied here:

But instead of passing a normal variable, we are passing a structure variable here.

passing the individual members

Let us understand this with an example program.

```
// defining a structure with 2 arguments rollno, percentage
struct student
{
  int rollno;
  float percentage;
};  // we have not declared any structure variables in this
    definition, we will declare the structure variables inside
    the main() function.

main()
{  // we are passing the members of structure as arguments to
// the function call, so the problem is reading the values to
// the structure members and displaying using the user defined
function, so the display function will be applied and the
display function is written and we call this in the main() function.
// so we have to declare the one structure variable like below.
  struct student s1;  // s1 is a simple structure variable which
// has to follow the structure of a student i.e. s1 is having a
// rollno and s1 is having a percentage.
// In order to access the members of a structure variable we
  require a dot operator.
```

// reading the values of rollno and percentage and it will be done inside the main() function.

printf("Enter rollno and percentage");
scanf("%d %f", &s1.rollno, &s1.percentage);

// now we have to print these values for this we can simply // write a printf function but here we will take one function which displays the rollno and percentage.
So here we will write a user-defined function with the parameters rollno and percentage and for this purpose we have to write a function call inside the main function.

display(s1.rollno, s1.percentage); // here i am not // passing complete structure i am just passing // arguments. so if there are 5 or 6 individual // members in a structure, we can pass only 2/3 members // or all it's upto the user but passing them individually.

actual parameters.

}. // close of main() function.

display(int a, float b).
{
// As it is call by value we have to take another variable so that the copy of these 2 members (called in main function) i.e. s1.rollno, s1.percentage will be assigned to the formal parameters in the function definition.

// just print the values of a and b.

printf("Roll No is %d", a);
printf("percentage is %f", b);
return;
}

passing the Entire structure:

In this instead of passing individual members, we can directly pass the complete structure i.e here it is S1.

```
struct student
{
  int rollno;
  float percentage;
};
main()
{
  struct student S1;
  printf("Enter rollno and percentage");
  scanf("%d %f", &S1.rollno, &S1.percentage);
  display(S1); // passing entire structure as parameter
}                to the userdefined function.
```

*** In function definition as we are passing complete structure, we have to declare another parameter i.e. a formal parameter which assigns the values of actual parameters. So here actual parameter is a structure variable and the formal parameter also should be a structure variable.

So here in function definition, we have to declare a structure variable as actual parameter is a structure variable.

So for creating a structure variable the syntax is:

```
struct structurename  structurevariable
```

So In function definition we write as follows:

P.T.O

display (struct student s2)
{
// This implies that s1 structure (the values of s1 structure are directly assigned to structure s2).
ie s1.rollno will be assigned to s2.rollno
s1.percentage will be assigned to s2.percentage

This statement represents `s2=s1;` i.e. s1.rollno s1.percentage values are directly assigned to s2.

// In this we are passing the complete structure as a parameter to the user defined function.

printf ("Roll NO is %d", s2.rollno);
printf ("percentage is %f", s2.percentage);
}

passing the address of a structure (call by reference).

Here instead of passing the parameter, we are passing the address of a structure.

```
#include<stdio.h>
struct student
{ int rollno;
  float percentage;
};
main();
{ struct student s1;
  printf("enter rollno and percentage values");
  scanf("%d %f",&s1.rollno, &s1.percentage);
```

```
display(&s1); // here we are passing address of the
                structure.
}
```

In display function definition we have to get the value at the address of s1 so for getting the value at the address we have to use "*" operator, so we have to create a pointer variable which refernces to structure

```
display(struct student *p)      <- here we have to create a pointer variable i.e. *p
                                   -> it is a pointer which points to the structure s1.
{                                  so the values are at *p.
printf("Roll NO is %d", p->rollno);
printf("percentage is %f", p->percentage);
}
```

Self-referential structures:

Definition:

Self referential structures are those structure in which one or more pointers point to the structure of the same type.

Ex:

```
struct self
{
  int p;
  struct self *ptr;
};
```

the above "struct self" is a structure which consists of two members, one is an integer variable 'p' and the other one is a

`struct self *ptr;` ⇒ this is the pointer to the struct self itself, i.e. why it is called self referential structure

→ Because here we have one or more pointers pointing to the struct of same type.

→ struct self *ptr; → it pointing to the same structure i.e. struct self. i.e. why the structure is called, self referential structure. Here we have a pointer which is pointing to the struct self) itself.

→ Now let us discuss it with an example.

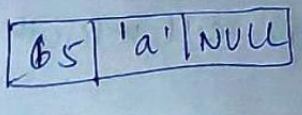
Example:

```
struct code {
    int i;
    char c;
    struct code *ptr; // ptr pointing to itself.
};

int main()
{
    struct code var1; // var1 is a variable of type struct code.

// var1 can access all the members of structure code without any problem.

    var1.i = 65;
    var1.c = 'a';
    var1.ptr = NULL; // Initially this is the case, i.e. it is containing NULL.
```

Variable var1 is accessing ptr which is a pointer to struct code itself but at the end it is a pointer it contains address.

Visualization:

Let us see how var1 will look like.



⇒ This is the visualization of a structure which consists of 3 members.

var1 is a variable of type "struct code" and it consists of 3 members and here in the diagram shown above it is initializing members.

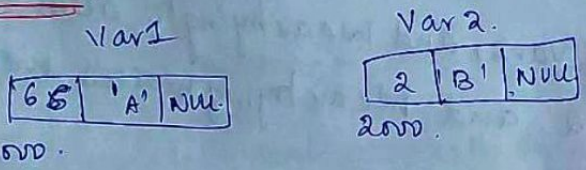
The first variable is initialized with the value 65
The second " " " " " " 'A'
The third " " " " " " NULL

and as the the third variable is a pointer, it has initialized with the NULL value.

Let's get back to the code and we will declare another structure variable.

```
struct code var2;
// with the help of variable var2, we will be initialize the
// members one by one again.
var2.i = 66;
var2.c = 'b';
var2.ptr = NULL.
```

Visualization:



Now let us suppose the initial addresses [starting addresses] Base addresses of these two structures are 1000 and 2000 this is our assumption of course, it can be anything. i am assuming these are my starting addresses.

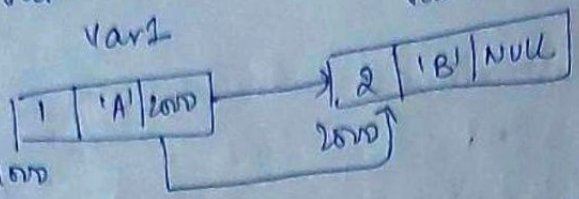
In the code i will add!

```
var1.ptr = &var2;
```

previously var1 address is initialized with NULL.
Now, var2 base address is initialized to ptr of var1.ptr.

Now var1.ptr will now contain the address of var2.

We know the base address of var2 is 2000, so ptr of var1 will contain 2000.



Now we can access var2 easily with the help of var1 as well Bcz var1 is having address 2000 which is the address of var2.

This means var1 has the capability to access var2.

We have represented this relationship with the arrow over here.

This means var1 can access var2 members as well That is why in printf function with the help of var1, we will try to access the members of var2.

```
printf("%d %c", var1->ptr->i, var1->ptr->c);
```

} Op: 66
B.

We are calling var2 here because var1 ptr contains the address of var2, and we are trying to access the member 'i' of var2. and we are trying to access the member 'c' of var2.

→ The above is the very important point!

With the help of var1 we are trying to access the members of var2.

Obviously the arrow operator is required because it's a pointer variable and with the help of this operator we can directly access the members of structure variable var2.

We can understand that we have declared 2 variables of type struct code, with the help of first variable, I am able to access the contents of the other variable as well and there is no problem at all.

→ Why this is possible, because here we have a struct code" which consists of a "pointer variable" (ptr) pointing to the "struct code" type of data only.

This is what about self referential structure.

How self referential structures are useful?

→ We will see the usefullness of self referential structures in linked list

Self referential structures in a better way with Example in C.

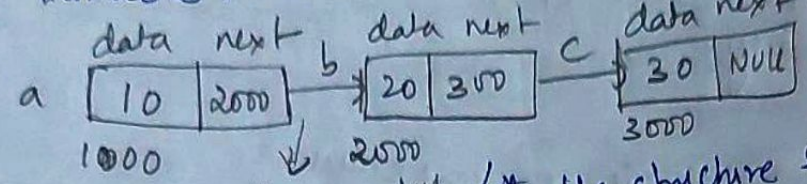
Defn:
A self-referential structure contains a pointer member that points to a structure of the same structure type.
It is used to create data structures like linked lists, trees etc.

Ex:
```
# include<stdio.h>
struct node
{ int data;
  struct node *next;
};
void main()
{ struct node a,b,c;
  a.data=10;
```

// structure is a user defined datatype and once we define a structure we can use as a datatype like int, float...
// self referential structures are generally used in linked list type programs.
here structure name is node you can give any name depending on program.

for structure members also you can give any name to the structure members.

→ once we define the structure tag we can use as a datatype and by using the data type we have declared 3 structure variables. and each variable takes 4 bytes of memory as every variable has 2 members



b.data = 20; this arrow tells the location of b is referred by a.next.

c.data = 30; /* the structure size we can calculate by adding the field. structure members required

a.next = &b; just like memory, data is an

b.next = &c; a linked list integer type so it occupies

// so here every structure variable 2 bytes / 4 bytes depending on the compiler.

takes for int = 4 bytes and and pointer of any type

let us consider for pointer it takes 2 bytes memory.

takes max of 8 bytes / so In turbo C, 4 byte on a

total 8 + 4 = 12 bytes is 32 bit machine and

allocated to each variable i.e. 8 bytes on 64 bit machine

a, b, c here.

and every variable tag have 2 members and in one member we can store integer i.e. data and in another member i.e. - next as it is pointer variable, we can store the address of any variable of this type of structure i.e. (struct node) here.

here we can store the address, 8 address / 2 address.

In the next member of 'a' address we can store even the address of itself i.e. a's address itself that is why it is a self-referential structure.

→ It is called self referential structure because it refers to itself, it stores the address of itself.

In the program we gave the data like;

a.data=10; b.data=20; c.data=30;

a.next=&b; b.next=&c; c.next = NULL;

printf("In the value of b=%d", a.next->data);

↳ (b.data) also we can write.

} but here i am accessing her in different way, i.e. through pointer i.e through address (for this we have to use →).

using structure member operator / dot operator we can access the integer data like a.data=10; ... so on.

→ But if you want to access a pointer variable data, we have to use → operator.

so in a.next we stored b address and in b.next we stored c address.

because. a address, b address, c address all these variables address is same as the struct node pointer type, so we can store any structure variable of this type can be stored in the next member here.

→ and it there is no next location i.e here after c we don't have the next location, so in c.next we placed NULL here.

In linked list each location is called node that is why we have given structure name as node here.

Unions in C:

It is almost same as structure but we have difference between structure and union

→ union is also a user defined data type like a struct

Let us understand all the below with proper Examples.

→ What is Union

→ How to define this datatype

→ How to use this in a program

→ Advantages of union

→ drawbacks of union.

→ Why we use union

→ How this is different from structure

What is Union?

Union is also a "user-defined datatype".

Structure is also a user defined datatype, which consist of different different elements which are having different different datatypes.

Syntax of structure:

```
struct student ← structure tag.
{
    int rollno;
    float marks;
};
```

These are members of structure

```
void main()
{
    struct student s1;
    s1.rollno = 50;
}
```

declaring a structure variable

→ Union is also a user-defined datatype (almost same like structure)

→ It also have information of different variables of different different datatypes.

→ Syntax is also same as of structure.

→ The only difference is what is ratter than struct keyword we use union keyword.

Declaration of Union:

Suppose for Ex:

```
union abc
{ int a;
  char b;
  float c;
};  ⇒ have to put semicolon here.

void main()
{ union abc u;
```

The declaration of union is exactly the same way as we do in structure except that instead of struct keyword we use union.

We can declare variable after definition of union like:

```
Union demo
{ int a;
  char b;
} u;
```

(or) we can declare globally like below.

```
union demo
{ int a;
  char b;
};
union demo u;
```

or we can declare union variable in main()

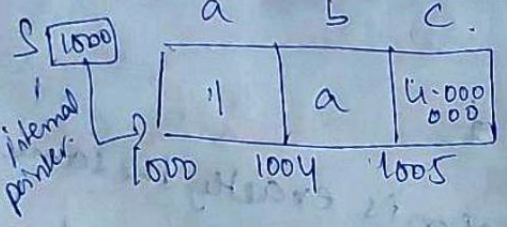
```
Union demo
{ int a;
  char b;
};
void main()
{ union demo u;
}
```

The only difference lies, in how the memory is allocated to the members of union and to the structure.

```
union demo
{
  int a;
  char b;
  float c;
};
void main()
{
  union demo u;
}
```

```
struct demo
{ int a;
  char b;
  float c;
};
void main()
{
  struct demo s=
  {1,'a', 4.0};
  printf("%d %d %f %f",
  s.a, s.b, s.c);
}
```

For a, b, c how the memory will be allocated in structure?



⇒ this is how the memory block will be there (contiguous memory locations)

Suppose base address is 1000, 4 bytes for int so 1004 for next memory location b. and for character only 1 byte for b and so c's memory starts at 1005 and total memory allocated for this structure is 4 for int, 1 byte for b & 4 for c = 9 bytes

and s is structure variable name and it will contain the base address, it is acting as internal pointer so it is pointing to 'a'.

In structure for each member, separate memory allocation is done.

How memory will be allocated to union members :)

In unions, only one memory block will be allocated.

Union → all the members of union will share a single memory

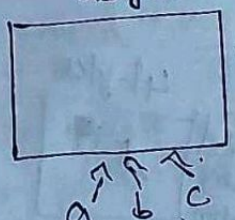
→ Now how many bytes will be allocated to this memory block :)

see in union defining we have given 3 members i.e. int a, char b, float c;

so generally int will take 4 bytes, char will take 1 byte, float will take 4 bytes.

→ The datatype which is having maximum memory allocation, that much memory will be allocated to the union.

so here



⇒ for a, b, c only 4 bytes will be allocated.
three variables will share the same memory allocation.

Accessing the members of union:

Accessing the members of union is same method as accessing the members of structure.

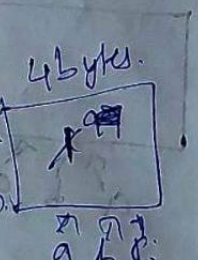
union variable dot operator union member

Union demo

```
{
  int a;
  char b;
  float c;
};
void main()
{ union demo u;
  u.a=1;  // accessing the members of union and initializing
  u.b=97; // Ascii value.  the values.
  u.c= 9.2;
 printf("a=%d", u.a);
 printf("b=%c", u.b);
 printf("c=%f", u.c);
}
```

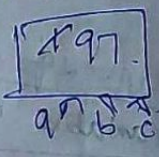
*** What will be printed here?

at First i am storing u.a=1 so 1 will be stored in memory location



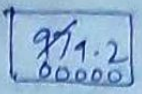
because 'a' is sharing the memory location.

→ next u.b = 97 and b is also sharing the same memory location, so the one will be replaced with 97.



4bytes.

⇒ now the c = 9.2; now 97 will be replaced with 9.2. as c is also sharing the same memory location.



So final value which will be stored in the shared memory location of a, b, c is 9.2.

> Whatever the value you entered at the last that value will exist in the memory, the previous values will be overwritten.

so if you print u.a → may be u will get some garbage value.
if you print u.b → may be you will get any garbage value.
If you print u.c → 9.2 it will print.

So the last value you entered that will be only stored.

> So this is the drawback of unions.

Why we use unions if we have drawbacks?

→ We don't use unions now-a days, unions were used many years ago and at that time "memory was very expensive".

→ If we see the structure is allocated 9 bytes and union is allocated 4 bytes.

→ unions will take less memory, so suppose a situation is we are having 5/10 members but at one time i want to process only one member

→ at one time suppose i want to process only `int a;` → this member i want to store the value in a and i want to access the value 'a', not in 'b' and 'c'.

so using of the structure will be wastage of memory. because here 9 bytes are allocated but we are not using 5 and variables and using only a.

This will be wastage of memory.

In this case union will be better.

In union, we cannot process all the members of union at the same time.

If you want to process one element at one time, in this case union is a good idea rather than using structure (it is wastage of memory).

- But Now a days, buying of memory is not so much expensive, memory is available at cheap prices.

→ so we don't care about the wastage of memory and we use structures only, we don't use unions.

only advantage of using unions is less wastage of memory.

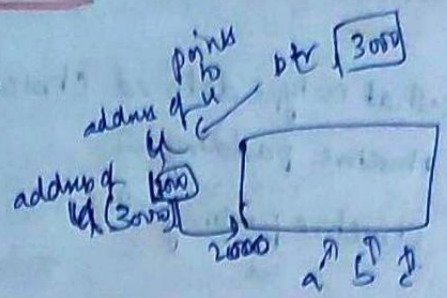
But draw back is, we can't process all the elements at the same time.

Because unions hold only the last entered value → because of the sharing of the memory (memory allocation by the members of union).

Syntax of structure and union will be the same.

using pointers also we can access:

```
union abc
{ int a;
  char b;
  float c;
};
void main()
{
  union abc u;
  union abc *ptr; // ptr is a pointer and we are assigning address of u.
  ptr = &u; // In "ptr" we are assigning address of u (3000 here suppose)
```



If you want to access the members of union using pointer variable, we have to use "->" (arrow) operator and process is same as that of structure.

```
  u.a = 1;
  u.b = 97;
  u.c = 9.8;
  printf("a = %d", ptr->a);
  printf("b = %c", ptr->b);
  printf("c = %f", ptr->c);
}
```

Every thing is same like structure, only difference is the memory allocation, how the memory allocation is done i.e. how the memory is allocated to the union members.

similarly we can use array of unions --- so on all, but the above concepts are enough for unions as we are not using unions these days.

⇒ Type defined structure:

What is type defined structure?
structure padding?
structure packing?

} These concepts are of structures

What is type defined structure?

How to define a structure using "typedef" keyword?

typedef is a keyword and it is used to create your own datatype

(or)

It is used to create "alias" of any datatype or the synonym. we can say of any datatype.

→ Like say you have two names, you will have your original name and your nick name also.

How to create an alias (or) a synonym of a datatype (or) how you will create your own datatype?

```
typedef int integer.
```

⇒ creating your own datatype

- typedef: use this keyword typedef
- int: old datatype
- integer: new datatype.

→ In your program you have to write like above to create our own datatype / alias to old datatype.

→ Suppose in your program, in main() function, I want to declare a variable of type integer so the datatype will be

```
int a = 10;
```

⇒ this is the syntax actually.

but if you have used `typedef int Integer;`

rather than `int a=10;` we can write as

`Integer b=15;` this is correct, b is a variable of the datatype int and its value is 15, because int and Integer both are same, Int is old datatype and is renamed as Integer.

Ex:
```
typedef int Integer;
void main()
{
  int a=10;
  Integer b=15;
}
```
// If you write like this, the scope of writing integer in place of int is in this program only we cannot use integer instead of int in another program without using typedef.

If you use typedef in functions, it will be within the function only.

Similarly we can use the typedef with the structure.

* Let us understand How we can use "typedef" with the structure and why we use this :)

What is the benefit of using this typedef :)

We will take the same example:

`struct student` → this is the name of the datatype.

```
{ int rollno;
  char name[20];
  float marks;
};
void main()
{
  struct student s1; // declaring a structure variable
  scanf("%d %s %f", &s1.rollno, &s1.name, &s1.marks);
  printf("%d %s %f", s1.rollno, s1.name, s1.marks)
}
```

Now We can see in the above way of defining and declaring a structure variable,

struct student ⇒ every time we create a variable of this datatype this structure datatype, you have to write the complete struct keyword followed by structure tag / structure name

ie. struct student ⇒ here.

This is little bit lengthy.

We can create our own name for this struct student datatype (this is old datatype, just like int).

So how we will create our own name for this structure datatype?

typedef struct student ⇒ old datatype.

```
typedef struct student
{
    int rollno;
    char name[20];
    float marks;
} student;
```

// this is not a variable here, it is alias / synonym for this datatype.

It is not variable, this will not take any space.

↳ this is new name of the old datatype

// Now if you want to declare a structure variable, rather than "struct student", we can simply write "student", that is also fine.

```
void main()
{
    student s = {10, "lakshmi", 90};
}
```

↓ this is variable.

But if you don't use typedef and write like:

```
struct student
{ int rollno;
  char name[20];
  float marks;
} student;
```

↳ this is just a structure variable → yes.

and now if you declare like:

student s = { 10, "Lakshmi", 90 }; ⇒ this will use wrong.

even if you declare this outside or inside of a main() function.

```
void main()
{ student s = {10, "Lakshmi", 90}; ⇒ this way of
                                     writing is wrong.
}
```

typedef int integer;

↳ these names also have local scope and global scope.

↳ if you declare this before the main() function i.e. globally then throughout the program, rather than int we can write down integer.

But within any function if you declare, then its scope is local to that function i.e. you can use only in that function.

We can use typedef and write many programs of structures in easier way.

p-T-O

⇒ Structure padding in C:

→ What is structure padding -?
(In interviews / gate / net we will get these type of questions.)

→ Why we need this -?

What is structure padding in C?

Ex:

```
struct demo
{
    char a;
    int b;
} s;
```

simple example of structure we have considered and here i am defining a "structure demo" and like how declared 2 members (char a & int b) and we have declared a structure variable s.

Now how much memory will be allocated to 's' or what is the size of the structure

we can say 1 byte for character datatype and 4 bytes for int.

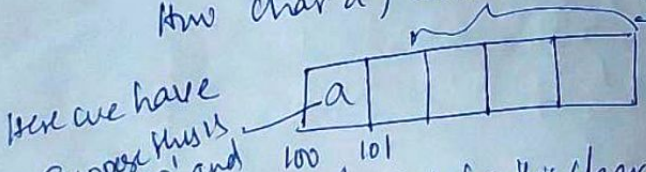
Let's suppose char takes 1byte, int takes 4bytes, float takes 4bytes, double 8bytes, long 8bytes ...

IMP So for the structure demo 1+4 = 5 bytes will be allocated to s -? No this is wrong. Why -?

How the memory will be allocated -?

Because of the allignment (the data allignment).

How char a; and int b ⇒ will be allocated same memory -?



Here we have suppose this is 'a' and address is 100 & 1 byte for this char

→ maybe you will say this 4bytes will be for int total 5 bytes - NO

Why so-?

Because, memory is not "byte addressable" now a days, it is "word addressable".

Word addressable means-?

↳ means, in one cpu cycle, at one time, the processor can fetch one complete word from the memory and what is that "word"-? (how many bytes this word will have-?).

on a 32 bit machine → How many bytes this word will have-?
word size is 4 bytes.

on a 64 bit machine → word size is 8 bytes
(processor)

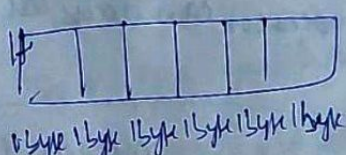
#this means on a 32 bit machine, processor can access 4 bytes at one time.

on a 64 bit machine - "processor can access 8 bytes at one time" from memory.

⇒ In one cpu cycle you can access 4 bytes / 8 bytes depending on machine & processor i.e. - this is the case.

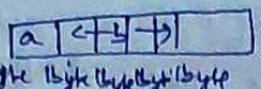
and historically:

processors were byte addressable i.e. processors at one time can access only 1 byte, then 1 byte then 1 byte --- soon.



But here the problem is what-?

that is taking 4 bytes and if processor is "byte addressable" then how many cycles (how many times the processor have to fetch the memory to read 1 integer value (4 times).



1time 2nd time 3rd time 4th time ⇒ these many times the processor has to fetch the memory to read this value, bcz int is 4 bytes.

Addresser can read only 1byte at a time

↳ So we require '4' cpu cycles.

Real time Example:

Suppose you want to buy shirt, Tshirt, pant, jeans.

at first time you will buy shirt and come to home

Second time you will go to shop and buy Tshirt and come & come to home.

then third time and 4th time you go to shop and buy other things.

Obviously this would be hectic.

What if we only go one time and pick all the 4 things, and come back to home, this is a good idea.

This is the same thing with "byte addressable" and "word addressable"

In byteaddressable → it is taking 4 cycles to read one integer variable.

What if "in one cycle only" that processor can access all the 4 bytes →

Yes this is possible, i.e. why now a days, memory is Word addressable. to increase the performance, to increase the speed. (at one time we can access these 4 bytes).

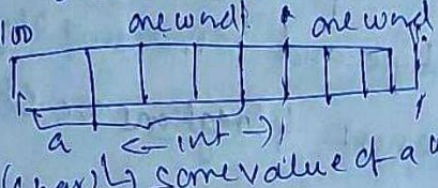
any one cpu cycle will be required and you can access 4 bytes.
in 32 bit.
If 64 bit, you can access 8 bytes. and in one cpu cycle only.

It depends on the architecture.

*** ⇒ Here i am assuming the word size is "4 bytes", means we can access 4 bytes at a time.

```
Struct abc
{
char a;
int b;
};
```

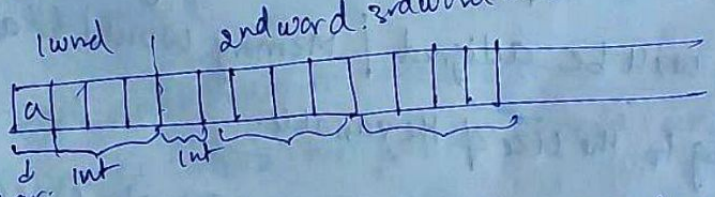
Suppose we have stored 'a' in memory, and it is of char type so it will take 1 byte. and in this 'a' we will have some value.



(char) ↳ some value of a we will store here.

and if we store the int (4 bytes) here, i.e. one word 4 bytes, then other word will be 4 bytes.

for int it will take 3 bytes after 1 byte which is stored for a and for b it will read 3 bytes of second and 1 byte is of another word size it will take.



Now if you want to access this integer value how many cpu cycles will be required → 2 cpu cycles.

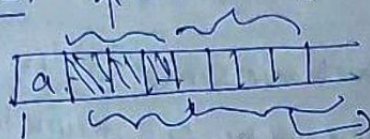
In one cpu cycle here we can access only 3 bytes and in another cpu cycle 4th a will access this 1 byte.

Obviously this processor will read complete word, but your information will be in First 1 byte of Second word for integer value. So 2 cpu cycles will be required means

yall can do this in one cycle but we are requiring 2 cycles because of the storage, we need 2 cycles.

i.e. **Wastage of cpu cycle** ⇒ which is going to decrease your performance

Second method:



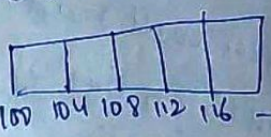
After this a value is stored i.e. 1 byte and remaining 3 bytes of the word is leaved and the next 4 bytes we store the integer value i.e. 'b' value.

Now how many cpu cycles will be required to fetch the integer value → **only one cpu cycle** because 4 are in one word location.

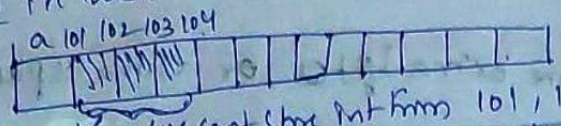
→ many processors expect that the memory of the structure members will be aligned / Memory would be allocated according to the size of the variable.

→ character is going to take 1 byte, so it is going to store any where.

→ But 'int' is going to take 4 bytes, so the address should start from "the multiple of 4" like you can say 100 4, 100 8, (or) 100 then 104 --- something like this



and here in this.



padding bits. we can't store int from 101, it is not a multiple of 4. (on 32 bit architecture)

we can't store integer value starting at 101

because it is not a multiple of 4, so you can start from 104 to assign int value and i.e. multiple of 4.

for char a; we have assigned 1 byte and left other 3 bits and these extra 3 bits which are left are known as padding bits / we can say memory holes.

(or) we can say memory allignment / data allignment.

(we are alligning the data just to increase the speed / or to increase performance of the cpu, but at a penality of memory (here 3 bytes is the wastage of memory).

But memory is now a days getting cheaper and cheaper, so we can attend some wastage of memory to increase the speed of the cpu.

→ If you don't want to have wastage of memory and memory is critical for you and speed is not, then you can avoid this padding.

But one thing, padding will be automatically done by the Compiler, you (u) will not do padding, Compiler by itself will do.

→ But if you want to avoid padding then you have to do packing, you have to include some special line in your program i.e. known as structure packing.

The process of inserting some extra bytes, or extra space between these variables, just to allign data, this process is known as "structure padding" inc.

So whenever you are asked how much memory will be allocated to the structure members →

Your first question will be what →

with structure padding / without structure padding.

(or)

with data allignment / without data allignment.

Next question should be →

What is the word size → 4 bytes / 8 bytes.

and on some compilers, int will take 2 bytes and on some machines, it will take 4 bytes.

If the interviewer doesn't tell you the details, you should ask the details of all these to get clarity and then you can give answer.

So now how many bytes will be allocated to the following structure →

```
struct abc
{
    char a; → for this 4 bytes.
    int b; → for this 4 bytes.
};
```

So total 8 bytes will be allocated

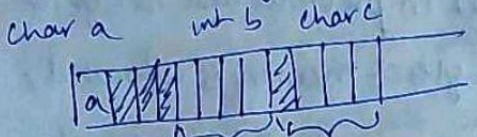
Another Example:

```
struct abc
{ char a;
  int b;
  char c;
};
```

for this now how many bytes will be allocated -> with structure padding ->

12 bytes.

The following will be the allignment.



⇒ char c takes 1 byte, but processor can access 4 bytes, this is word addressable, so the complete word i.e. 4 bytes will be for char c" account (not 1 byte only)

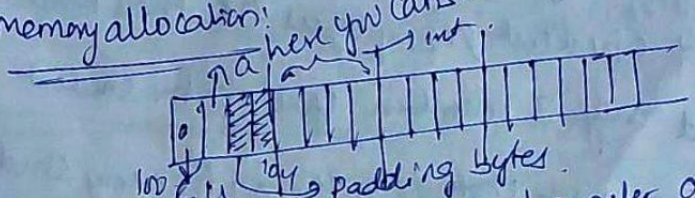
So the memory allocation will be for 3 members i.e. 3×4=12 bytes of memory will be allocated.

Another Example:

```
struct abc
{ char c;
  char a;
  int b;
};
```

if you define like this, we wrote char c before char a then int b, Now how many bytes of memory will be allocated now ->

memory allocation:



allocated 1 byte as it is character and next is char a, and character is going to take only 1 byte, so you can store it any where -> so now the next 1 byte after c can be allocated to a.

and next is int. and we cannot store 2 bytes of int in 1 word and 2 bytes in another word.

int will be stored from 104. to 107.

How many padding bytes are there here -) only 2 bytes.

Total memory allocated here will be 8 bytes.

→ In the previous case also we have 2 char and 1 int, here also we have 2 char and 1 int but in previous case 12 bytes of memory was allocated, here 8 bytes of memory is allocated,

→ now a days the processors are very smart, they can handle even these kind of allignments, this problems come when you are using "risc" type of processor, they throw the exception when, data is not alligned in the memory. (processors will handle)

→ Sometimes in very worst situations, maybe cpu have to use 2|3 cpu cycles to access some data, because of the bad allignment, so it is the duty of the programmer to take care of these things, you have to write a program in such a way that memory wastage will be less.

→ If you are storing large amount of data suppose, and structure is generally used for what -) to store the information of student, employees or there kind of things and any 3 members you take for student

for ex: and if 1000 / more students are there, for every student we will be wasting 4 bytes if you don't write the program properly, Suppose instead of adhaar card you take - more wastage of memory if you don't write program properly. (Programmer has to declare members of structures in a proper way)

1. Declare structure members / write the structure members in the increasing order of their size (memory size), so that memory size can be properly utilized.

Every datatype in C will have alligment requirement (intact it is mandated by processor architecture, not by language).

→ padding bits contain one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. Which is called structure padding.

Structure packing in C:

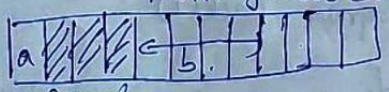
What is structure packing-?

Without structure padding
char will take 1 byte.
int will take 4 bytes
total 5 bytes.

```
struct demo
{
    char a;
    int b;
};
main()
{ struct demo s;
  printf("%d", sizeof(s));
}
```

But as compiler inserts automatically the structure padding for data alligment.

memory allocation using structure padding for struct demo



this is wastage of memory.

o/p
8 bytes.

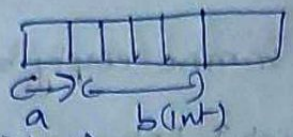
We can see to increase the speed or efficiency of cpu cycles / speed of execution we can say for structures we use structure padding but under the penality of wastage of memory.

If i don't like to waste the memory -)

We would like to pack everything.

We don't want the compiler to insert any extra bytes in between the variables, don't want any allignment of the memory.

like i want like below



total i want 5 bytes of memory only.

I want to pack the structure. ⇒ This is what we call it as structure packing.

What we have to do -)

We have to add a simple line in the program after the header files

pragma pack(1)

↳ it is a directive which is used to turn on and turn off some features in your program, here we are turning on the packing features.

pack(1) ⇒ you have turned on the packing here, So that means you have turned off the padding, now the compiler will not insert these extra bytes.

This is what is structure packing. you can save lot of memory with the packing but at a cost of performance (more cpu cycles).

pragma pack(1) ⇒ will tell the compiler to pack the structure members with a particular alignment.

particular alignment → means if you don't want to give, you want to give 2/4 ⇒ you are providing your alignment (upto how many bytes can be padded).

Imagine 1 byte ⇒ no padding will be there ⇒ 1 byte alignment will be there, byte by byte alignment will be there to access the memory. (processor can access the memory).

```
#include <stdio.h>
struct demo
{ char a;
  int b;
  char c;
};
void main()
{ struct demo s;
  printf("%d", sizeof(s));
}
```

o/p: 12 bytes.

```
#include <stdio.h>
#pragma pack(1)
struct demo
{ char a;
  int b;
  char c;
};
void main()
{ struct demo s;
printf("%d", sizeof(s));
}
```

o/p: 6

Assignment to students:

```
struct demo
{ char a;
  int b;
  double c;
};
```

How many bytes will be allocated with structure padding, without padding and with packing →

Bitfields in C:

Bitfields is very important topic, in case you want to optimize your memory we use bitfields.

What exactly is a bit field?

Let us take an example:

We have a structure demo

```
struct demo
  {
    int Member1;
    int member2;
  } d;

int main()
  {
    printf("%d", sizeof(d));  ⇒ it will give 8 (8 bytes)
  }
```

There are no other datatypes, so there is no question of padding otherwise there is a chance of padding, so minimum will be 8 bytes and it can be more sometimes.

Now what we will do is-?

```
#include <stdio.h>

struct
  {
    int member1;
    int member2;
  } d;
```

} i didn't give structure tag.

```
struct
{
    int member1 : 1;
    int member2 : 1;
} d2;
```

In this after declaring a member of structure int member1 i have given : (colon) then 1 and after int member2 also i have given : (colon) then 1 → what does this mean?

```
int main()
{
    printf("size of Regular structure = %d\n", sizeof(d1));
    printf("Size of structure with bitfield = %d", sizeof(d2));
}
```

Op:

Size of Regular structure = 8.

Size of structure with bitfield = 4. ⇒ How come?

Explanation:

→ When you are putting colon 1 (: 1) ⇒ it says that after the member we have given :1 ⇒ this says that this member is going to have the value as either '0' or '1' ⇒ nothing else.

VVV Imp → When we are very sure that this member which we gave colon 1, is not going to have all integers, we can restrict the size of this to a particular no of bits, so rather than giving 4 bytes for integer, which happens in the first structure we declared above in the program, and in the second structure, in the program, for member1 in second structure, for int ⇒ only 1 bit is reserved and for member2 of second structure also ⇒ 1 bit is reserved.

If we change the second structure as below:

```
struct
{
  int member1;
  int member2;
} d1;

struct
{
  int member1 : 10; } // after colon instead of 1 gave 10 in both
  int member2 : 10;
} d2;

// Now printing the size of structures.

void main()
{
printf("Size of Regular structure = %d", sizeof(d1));
printf("Size of structure with bitfield = %d", sizeof(d2));
}
```

Op: Size of Regular Structure = 8. (Now the size is 4 — How?)
Size of Structure with bitfield = 4.

Explanation:

If you colon: 1 or colon: 2 ---- colon: 32 for a member in a structure, we will not have problems,

But change like:

```
struct { int member1 : 16;
         int member2 : 16; } d2;
```

and print size it will also give 4 as structure size

// Now we will change like below.

```
struct
{ int member1 : 16;
  int member2 : 17; // here it is taking 17 bits.
};
```

16 + 17 = 33 (beyond 32.)

So it will now print size as 8.

It will take in the multiple of 4 bytes the memory will be allocated)

// Now we will change like below.

```
struct
{ int member1 : 1;
  int member2 : 1;
  int member3 : 1;
  .
  .
  .
  int member32 : 1;
};
```

until 32 size it will take size of 4 for this structure.

↳ but if you add one more member then the size will be taken in multiple of 4's and size will be considered as 8.

What is the advantage of the bitfield?

→ The Basic advantage of the bitfield is, we can optimize the usage of memory, instead of taking sum of all the members of a structure based on their datatypes, we are not taking sum of all bits, and almost rounded to the nearest multiple of 4 bytes (integer).

VVIMP

⇒ But there is a point we need to take care of, some precaution we need to take care of.

Let us say:

```
struct
{ int member1 : 3; // 3 bits we have taken here
  int member2 : 1;
} ds;
```

What does 3 bits mean-?

3 bits basically means, we can have a binary value maximum of 111 ⇒ i.e. 7 only.

Suppose the above member1 has the value greater than 7

We will see that immediately the value is going to be lost.

Example:

```
struct
{
  int member1;
  int member2;
}d1;
struct
{ int member1 : 3;
  int member2 : 1;
} d2;
int main()
{
```

printf("size of Regular structure = %d \n", sizeof(d1));
printf("size of structure with bitfield = %d", sizeof(d2));

d1.member1 = 7;
d2.member1 = 7;
printf("%d %d", d1.member1, d2.member1);
}

It prints 8 4 7 [-1] ⇒ y it became -1 bcz

The last bit is going to be reserved for sign bit.
and at sign bit, when I say 7, the value which has comes
1 there.

Now we will change it to unsigned.

---

struct
{ unsigned int member1;
unsigned int member2;
}d1;

struct { int member1 : 3;
int member2 : 1;
}d2;

int main()
{ d1.member1 = 7; d2.member1 = 7;
printf("%d %d", d1.member1, d2.member1);
Otp : 7 7.

d1.member1 = 8; d2.member1 = 8;
// here in d1 we will not have any problem, because member1
is going to use all 32 bits and 8 basically in binary is 1000
but in d2 structure for member2 we have reserved only
3 bits.

" " " "
only in 1000 ⇒ 000 only will have the value and
this 1 will be truncated. so for d2.1 now it is
going to print 0 0 */

printf("%d %d", d1.member1, d2.member1);
Op: 8 0
}

→ We have to use bitfields, only when we are sure that in so many number of bits, the value which you are going to accomodate can be stored. and,

→ you have to ensure that after giving bitfield, the value should be accommodated in so many bits (not exceed that many bits (1000 for Ex).

→ This is the extra precaution you have to take when using bitfield.

But at the same time, the advantage of bitfields is Better usage of memory (only to the extent we are allocating the memory, not extra).

yes this is an Extension on structure

This facility is only for structures and unions. We can make use of it.

But union → it doesn't become useful, because, union any how will take the memory of largest member in the union

So bit fields are useful only in structures.

Note: The number of bits in the bit-field.
The width must be less than or equal to the bit width of the specified type.

⇒ Command line arguments:

What is an argument?

argument is an input value or an input element.

What is command line?

It is nothing but, "Command User Interface" (or) we can call it as "Character User Interface".

The best example for this is DOS Operating System.

The blackscreen / command prompt simply.

What is the meaning of command line arguments-)

Generally we can write, compile and execute the C programs using IDE (Integrated Development Environments like TurboC, Codeblocks, VS code editor --- soon).

Here we are discussing about DOS Operating system.

What is the use of DOS operating system-)

Generally, whenever we install any OS in your computer, two types of OS will be installed (one is CUI, one is GUI).

CUI ⇒ Command User Interface.

GUI ⇒ Graphical User Interface.

→ Mostly we all or we can say the end users use the Graphical User Interface.

→ The character User Interface is used by the programmers

OS is of two types as told i.e. CUI & GUI.

→ Generally as a programmer, We always work with CUI only because it is easy to write with CUI (people think we have to learn commands of DOS).

Command user Interface:

If you want to compile the program after writing the program, and suppose there is no IDE to write a program then how to write the program?

Steps of executing a program if you are not using the editor.

→ 1. Open Notepad and write program.
2 - Save the program with .c extension.
3. We have to open DOS (Command prompt) and locate the compiler.

Compiler is a .exe file on windows OS.

Example: Turbo C compiler ⇒ tc.exe compiler.

With the help of compiler only, you have to compile the application.

4. Compile program name.c / Compile source program then it will generate the compiled file.

5. Run program.

The above is a bit difficult of writing, locating compiler, compiling and executing.

But the above is the exact process of writing a c program / for developing 'c' applications or any other c++ applications...

In turboc ⇒ for saving ↳, Compiling Alt+f9, Executing → Ctrl+f9 and o/p ⇒ Alt+f5.

↳ But the background process what happens is above steps only.

one should clearly understand the process of writing, compiling and Execution of a program i.e. why we have to follow the DOS OS.

Commandline arguments!

Whenever we are compiling and executing the applications, How to pass arguments to the program from the command line is called command line arguments.

How to pass the Commandline arguments-?

cmd/>

The above is Command prompt.

Let us say you want to compile a program. and program name is Let us consider as sample.c

With the help of compiler (tcc.exe) we have to compile

$ sample.c

```
cmd/> tcc.exe program.c
```

After compilation, what files will be generated?

program.obj files will be generated.

program.exe files will be generated.

How to Execute?

```
cmd/> program.exe
```

↳ how to pass commandline arguments now.

cmd/> program.exe arg1 arg2 arg3 arg4 ... (space should be given)

No of arguments can be passed (no limitation)

↳ followed by this we have to pass the arguments.

↳ these are input values (means arguments).

from the commandline we are passing (arguments) i.e. input values to the program.

Who is responsible for collecting this arguments & how to process all the arguments and what type of data / arguments we can pass?

How memory will be allocated to commandline arguments?

once you pass the commandline arguments - then the memory will be allocated to the command line arguments.

→ How to compile a c program manually?

from the command prompt / from the os environment we will with help of compiler manually is:

cmd/> tcc program.c <-- (click key Enter) → compiles

cmd/>. program (if this is how we execute the program without arguments) → execution without arguments.

But if you pass the arguments, followed by program name you can do like below.

When

```
              int  short|char  double|float  string
cmd/> program 10   g           34.56         clanguage
```

↳ 'n' no of arguments you can pass at the time of execution followed by the program name.

→ command line arguments is of type "string type"

Command line array is a string type because.

main (char* argv[].)

↳ main function is taking character pointer argv[])

↳ this is used to read command line arguments.

and the arguments mentioned above in example Ex:

10, g, 34.56, clanguage are "string type", so,

the memory will be allocated for this [10] [g] [34.56]
int ↓ character ↓ float.

[clanguage] (in different memory locations. all these gets memory allocated).
↓ string

What is the size-)

Here in C language it is very important.

cmd/> program 10 g 34.56 clang.

↓ the program name is also treated as 1 argument.

so all these 5 gets memory allocated at 5 different locations.

suppose simply we take address as 100 (base address by program

10 ⇒ address is 200

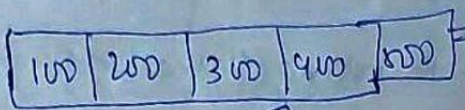
9 ⇒ address is 300

double ⇒ address is 400
34.56

clang ⇒ address is 500

like this 5 strings get memory allocated at 5 different locations.

and all these strings will be stored into 1 array.



⇒ all the 5 arguments will get the memory allocation in this array.

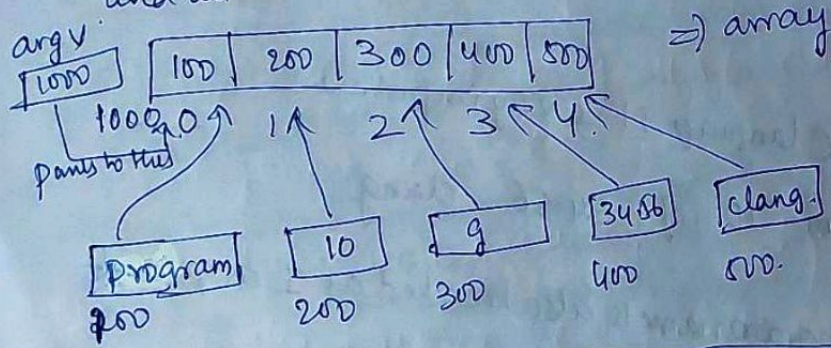
and memory allocation like this all will get

Q, 1⇒) How many arguments we have passed and how we can collect these-?

for these there is a variable called "argc" ⇒) it will maintain the count of no of arguments you have passed including the program name also.

so "argc = 5"

and all



⇒ array also contains one base address

so, let us say base address of this array is 1000.

this base address of array will be stored in variable called argv | 1000. ⇒) In C, array variable holds base address of the array.

so this is how the memory will be allocated to the command line arguments.

Accessing the values:

⇒ If you want to access argv[0] ⇒ it will give program name.

⇒ If you want to access argv[1] ⇒ it is first argument it will give 10 and so on. till index 4 (5 values) we can access.

VV imp

and here if you forcefully try to access argv[5], that is the value in this location it gives NULL pointer. will be present.

↓ here we don't have that index

Why NULL pointer?

Because this array is "pointer type array" / "character pointer array".

How to Execute programs using command line arguments-?

Simple program: hello.c

```
#include<stdio.h>
void main()
{
}printf("hello");
```

o/p: hello.

compilation at command prompt

go to c:\turboc2>.

tcc hello.c

Execution:

c:\turboc2> hello

just the name.

Another program: Sample.c

```
#include <stdio.h>
main(int argc, char *argv[])
{
printf("Argument count: %d", argc);
}
```

⇒ Here we are writing logic for only argc, but we have not written any logic for argv variable so compiler when compiles gives you a warning that "argv" was never used ⇒ this tells that we are wasting the memory.

C:\turboc2> tcc Sample.c

C:\turboc2> Sample.

olp: Argument count: 1 ⇒ here we didn't pass any arguments but still it gives 1 because, program name itself is one argument.

C:\turboc2> Sample 10

olp: Argument count: 2

C:\turboc2> Sample 10 g 34-56 dang...

olp: Argument count: 5

This is how we will pass arguments.

VIMP ⇒ Now after passing the arguments, I want to collect them in application and print the values — how?

program: Sample1-C

```
#include<stdio.h>
main(int argc, char *argv[])
{   int i;
    if (argc==1)  // only program name
      { printf("No elements to display----\n");
      }
    else
      { printf("List of elements: \n");
        for(i=1; i<argc; i++)
          {
            printf("%s\n", argv[i]);
          } // argv[] => array is of type string type and
// in C language we display string elements using %s.
      }
}
```

Compiling: C:\TurboC2> tcc sample1-C.

Execution: C:\TurboC2> sample.

No elements to display (we didn't pass any arguments. Program name itself is an argument).

Execution: C:\TurboC2> sample 10 9 34.56 Clanguage.

List of elements:

10
9
34.56
clanguage.

This is how we process Command line arguments from C application.

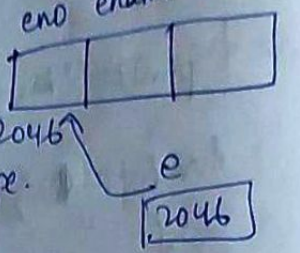
⇒ program to initialize a structure by reading elements from Command line.

How to pass elements to a structure / how to initialize a structure by reading input from the command line.

program: sample2.c.

```
#include <stdio.h>.
Struct employee
{
  int eno;
  char* ename; // pointer type.
  int sal;
};
int main(int argc, char *argv[])
{
  struct employee e;
}
```

// whenever you are allocating memory, structure variable holds base address only



// how to access structure elements →

// If you want to store information into these locations and if you want to access the locations we use "dot operator" ↓ accessor.

⇒ Now i don't want to use scanf function or any other function and i want to read the values from commandline

// Whenever in a structure here in "struct employee", first Whenever we are collecting the values, how we pass the arguments from the command line →

compilation: C:\> cd turboc2

C:\turboc2> tcc sample2.c

C:\turboc2> sample2 109 Lakshmi 30000

argv[1] argv[2] argv[3]

here argv[0] ⇒ don't need conversion because by default it is string type

but argv[1] and argv[3] ⇒ we have to convert to integer

So what we have to do ⇒ we have to use atoi() method.

program: sample2.c

```
#include<stdio.h>
struct employee
{
  int eno;
  char* ename;
  int esal;
};
int main()
{ struct employee e;
// we cannot write like e.eno = argv[1]; ⇒ e.eno is
// an integer type and argv[1] is a string type. So we have to write
// as.
  e.eno = atoi(argv[1]);   // we are collecting the input
                           from the command line and
  e.ename = argv[2];       we are storing into
  e.esal = atoi(argv[3]);  locations eno, name and salary.
  printf("Eno = %d Ename = %s Esal = %d",
          e.eno, e.ename, e.esal);
  return 0;
};
```

Compilation: C:\turboC2> tcc sample2.c

C:\turboC2> it will give warning: parameter 'argc' is never used in function.

because we have not used argc in our program.

execution: c:\turboc2> sample2 101 Lakshmi 30000

eno: 101
ename: Lakshmi
salary: 30000

We are passing input from the command line while executing the program and the values collected are stored in corresponding structure members locations according with help of argv[] array by performing conversions and we are printing the values through the program.

Program to read command line arguments and how to process Commandline arguments.

⇒ Suppose i want to perform addition operation, i will take 2 values (integers) from commandline and i want to perform addition operation and display

Remember: Command line arguments is a string type.

But here i will take integer values.

→ so we will pass integer values at the command line but internally that integer will be converted into string type.

How can we convert the string datatype into the integer type. for this we have to include "dos.h" header file.

dos.h ⇒ file is providing two functions to convert "string to integer" data and as well as "string to float" data.

Function to convert string to integer:

atoi => function

atoi(char *s); => it will take string as an argument and after conversion, it will return integer value.

Function to convert string to float:

atof(char *s); => it will take string as an argument and after conversion, it will return float value.

These are the two functions available in a command line.

program: sample3.c

```
#include <stdio.h>
#include <dos.h>
int main(int argc, char * argv[])
                    ↳ this is used to collect commandline arguments in c.
{
// read 2 integer values from commandline.
// i.e. cmd1> Sample3 10 20  => like this at command line.
                argv[0]  argv[1]  argv[2].

// To perform addition operation we need minimum 2 arguments.
// here what is the argument count here -> it is greater than
                                (3)          2

// so it starts with 3.
    if(argc > 2)
    {

    }
    else { printf("Insufficient input values");
           } return 0; }.
```

Compilation: tcc Sample3.c.

C:\turboc2> Sample3

Insufficient input values.

C:\turboc2> Sample3 10

Insufficient input values ....

C:\turboc2> Sample 10 20 ✓

but it will not give output bcz we didn't write the logic still.

```
#include<stdio.h>
#include<dos.h>
int main(int argc, char* argv[])
{
  if(argc >2)
  {
    char *s1 = argv[1]; // we are declaring character pointer variables s1 & s2
    char *s2 = argv[2];
  }
  int x = atoi(s1); // taking strings as an argument & returns integer.
  int y = atoi(s2);
  printf("sum = %d\n", x+y); }
  else
  {
    printf(" Insufficient input values");
  }
}
```

strings ⇒ means character pointer.

and we are collecting values of argv[1] and argv[2] (strings) and storing them in s1 & s2

Compilation: tcc Sample3.c.

Execution: Sample3.

Insufficient Input values.

C:\turboc2> Sample3 10 20

sum: 30.

C:\turboc2> Sample3 4 5

sum: 9

suppose you give

c:\turboc2> sample3 10 20 30 (gave 3 arguments)

No error. but it will take only first 2 values and prints sum of these values.

Sum: 30

⇒ c - preprocessors.

preprocessor:

The c preprocessor is not a part of the compiler, but it is a separate step in the compilation process.

> "c processor is just a text substitution tool and they instruct compiler to do required pre processing before actual compilation".

⇒ All preprocessor commands begin with a pound symbol (#) hash.

⇒ Basically whenever you use HashSymbol (#), it represents a preprocessor and preprocessor means, that process will happen even before the compilation i.e. pre compilation part.

⇒ so at that particular time, the compiler will not be working and whatever will happen will be taken by the time of compilation.

⇒ something which happens before the actual processing starts is caused by the preprocessors.

List of Important preprocessors.

Directive	Description
# define	Substitutes a preprocessor macro
# include	Inserts a particular header from another file.
# undef	Undefines a preprocessor macro.
# if def	Returns true if this macro is defined.
# if n def	Returns true if this macro is not defined.
# if	Tests if a compile time condition is true.
# else	The alternative for #if
# el if	#else and #if in one statement.
# end if	Ends preprocessor conditional.
# error	prints error message on stderr
# pragma	Issues special commands to the compiler, using a standardized method.

define:

- It is used for the macro
- We can also use this # define for containing a constant value.

include

→ We will use this in every other program of C.

like #include <stdio.h> --- so on.

→ for including any header file into your program we use #include.

→ so whenever your file is included, that #include line will be replaced by all the text which is there in the header file.

#undef:

If you have already defined something then later you can undefine that using #undef.

#ifdef and #ifndef ⇒ returns true if macro is defined or if macro is not defined.

#if If you want to go for a particular condition, before the actual program starts i.e. you want to check some conditions before actual program starts then we go for #if

→ suppose i want to check on which system you are working, i have written a code and for different OS's i have written different header files for that.

→ so if i have to check that before the actual programming starts using #if, #else, #elif and #endif are used.

#error: prints the error message if there is an error.

file have some predefined pre-processors here.

lets have the look on how we can implement them.

p-T-O

Below are some preprocessor examples.

```
#define maxarraylength 20
```
→ value
↳ name of macro

```
#include <stdio.h>
#include "myheader.h"
```
↳ header file. (we can include using angular brackets or through double quotes)

```
#undef FILE_SIZE
#define FILE_SIZE 42
```

```
#ifndef MESSAGE
#define MESSAGE "you wish!"
#endif
```

→ If you use angular brackets for header files, that means your header file must be at the same location, which is being defined by your environment

— Every Editor of C program has an environment where you can set the location (the default location of the header file)

— But if you want to use any customized header file in your program then you can use double quotes along with the absolute / relative path.

⇒ If you want to undefine like let us say we have defined max array length as 20.

⇒ You can just use undef filesize. and again define. size again.

#ifndef => If not defined means here then is a macro which is not defined if not defined then define that macro.

> There should not be multiple definitions of the same macro

predefined macros

The predefined macros should not be directly modified

Macro	Description.
__DATE__	The current date as a character literal in "MMM DD YYYY" format.
__TIME__	The current time as a character literal in "HH:MM:SS" format.
__FILE__	This Contains the current filename as a string literal. (gives the information of the current file).
__LINE__	This Contains the current line number as a decimal constant. (Tells the current line in which particular macro is used).
__STDC__	Defined as 1 when the compiler compiles with the ANSI Standard. → which ANSI standard is used.

```
#include <stdio.h>
main()
{ printf("File is %s\n", __FILE__);
  printf("Date: %s\n", __DATE__);
  printf("Time: %s\n", __TIME__);
  printf("line: %d\n", __LINE__);
  printf("ANSI: %d\n", __STDC__);
}
```

⇒ The C preprocessor offers following operators to help in creating macros: (which are associated with preprocessor directives)

- Macro continuation (\)
- Stringize (#)
- Token pasting (##).

Macro continuation:

like if you have something / some values, which you want to split in multiple lines then we can use this slash symbol (\) for the continuation.

Macro continuation (\)

→ A macro usually must be contained on a single line.

→ The macro continuation operator is used to continue a macro that is too long for a single line.

```
#define message_for(a,b) \        ⇒ first line we have named the macro.
printf("#a" and "#b": Welcome! \n")
```

↳ in the second line we have the value for it.

for that i can use just '\' and i can break the code in a separate line.

actually you can write the above complete thing in one line.

P.T.O

Stringize (#)

→ The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant.

→ This operator may be used only in a macro that has a specified argument or parameter list.

```
#include<stdio.h>
#define message_for(a,b)\
printf("#a" and "#b": Welcome!\n")
int main(void)
{
    message_for(Tom,Jerry);
}
```

symbol is called stringize, even we have used in macro continuation.

For what it is used for-)

It is used for converting the macro value into a string

like #a and in a value Tom will go and in #b, b value will get Jerry.

P.T.O

Token pasting (##)

The token pasting operator (##) within a macro definition combines two arguments.

It permits two separate tokens in the macro definition to be joined into single token.

```
#include <stdio.h>
# define tokenpaster(n) printf ("token" #n "= %d", token##n)
main (void)
   {
     int token34=40;
     tokenpaster(34);
   }
```

Implementation of macros & preprocessor directives and the operators using a program

```
#include "stdio.h"
#include <stdlib.h>
# ifndef MESSAGE
# define MESSAGE "In welcome to ---- In\n"
# endif
# define message_for (a,b)\
printf (#a " " #b ": simple easy learning !\n")
# define tokenpaster(n) printf ("token" #n "= %d", token##n)
int main()
  { int token34= 40;
     printf ("file : %s\n", __FILE__);
     printf ("( Time: %s\n", __TIME__);
```

```
printf("Line : %d\n", __LINE__);
printf("ANSI : %d\n", __STDC__);
printf("%s", message);
message_fr(TOM, Jerry);
token paster(34);
return 0;
}
```

olp: FILE: Main.c
TIME: 18:56:43
LINE: 16
ANSI: 1
Welcome
Tom Jerry : Simple
Easy learning!
token34 = 40.

Explanation:

→ In the implementation of pre-processor directives, you can see, we have started with "#include".

→ actually we start, the C programming code with this particular line (#include) itself.

→ We include the header files, for getting the prototypes of the methods (predefined functions).

→ We can see that we have included header files in two different ways.

In the first way:

1. We have included the name of the header files in double quotes

In the second way:

2. We have included the name of the header file in angular brackets.

→ In both the cases, by default the header file will be searched in the default location (depends on the environment) on which you are working.

But if there is any user-defined header file which is possibly in some other location, in that particular case,

- You cannot go for this angular bracket, rather, you can go for double quotations inside which you can pass the relative / absolute path of that header file.

- We have used some other preprocessor directives like,

```
#ifndef MESSAGE
#define MESSAGE "\n welcome in---- \n\n"
#endif
```

} ⇒ what are these.

Basically if you want to define some macro and if you want to check whether that macro was existing or not (earlier), for that

→ we have written #ifndef ⇒ which means if not defined MESSAGE i.e. if MESSAGE macro is not defined then in that case, I have defined that with message welcome... so on.

→ so Basically what will happen is, whenever you use this MESSAGE anywhere in your C program, that complete message or that complete macro name will be replaced by this

welcome---- value.

→ Basically when the compiler will compile the code, it will get something like "MESSAGE" written inside printf.

→ because what macro does → It will just contain a value and before the actual compilation starts

that macro name will be replaced by the value which we have set inside it.

Similarly how(# define message_for(a,b) \) ⇒ here we are going to pass two parameters in another macro called message_for a and b, ⇒ so here we have also used the "stringize" means i am going to convert that to a string, the values and i am using the continuation operator(\) because the name of the macro and its value are in the different places.

→ What we have done in the program is: there is (~~message for~~ - message_for(Tom, jerry); ⇒ here you can see, Tom and jerry.

→ There are two parameters 'a and b' and in 'a' i have assigned Tom and in b, i have assigned jerry

→ Since we have not enclosed Tom and jerry in double quotes here, so i am just using the hash operator before a and b, so that the values will be like string by the time it will be printed.

Before the above thing, we have printed some pre-defined macro's like:

getting the file name, current Time, line no like where particular macro is written in the program, and STDC the compiler version.

We can see one more concept here, where we defined the

```
#define token paster(n) printf("token" #n "=%d", token ## n)
```

↳ We have a macro here which is having a name token paster and a parameter that it will take (n) and printf ("token" #n "=%d", token ## n)

↳ you can see like this one is the value we pass here.

But here itself, we have. What is that (token ## n) ⇒ this is expecting some variable of the name, for example if i am passing "34" here then this n will be containing 34 and it will be looking for a "variable" with the name token34 and as you can see in the program we have already defined the value as 40 for this variable name (token34).

⇒ So when we execute, token34 = 40

↳ is printed as a string but this particular name token ## n name would also required.

If i will not define this variable or if i define the same variable with some other name it will give an error (like instead of token34, it will give token24 because i passed here 34.

like if i do something here itself then it will be fine because the variable name is treated as token24 which we have declared.

This is how we can start with basic implementation of preprocessor

Enumeration in C (enum in C).

→ enum is a user-defined datatype.

→ As we have Structure and union, same way we have enum to define our own datatype.

→ How to use "enum" to define our own datatype.

→ How to declare a variable of "enum" type

→ How to use enum in our program.

→ Important points of enum datatype.

→ Why to use enum.

→ What is the need of enum.

→ How this is different from "macros" and how this is different from "structure and union".

What is enum?

It is a user defined datatype and it is basically used to assign names to integral constants.

Example: `# define MAX 50` we know this.

and in whole program (or macro) whatever we use this MAX variable it will be substituted with 50, means we are we are using/changing the name, rather than this 50 and replaced it is an integral constant, but we are assigning a name to this integral constant. and the name is MAX.

and this name can be used throughout the program.

Same thing this "enum" will do, but what is the difference? Why we use enum over macros?

→ Sometimes we use macro's.

→ Sometimes it is better to use enum.

→ But the meaning of enum definition is:

"it is used to assign names to integral constants"

```
# define MAX 50
```

↳ this is an integral constant, we are assigning a name to it and we are using this name to our program.

→ When we use names,

- It is going to increase the readability of the program
- In my class let us say roll no 45 standup, and roll no 45's name is "Rahul" suppose.
  → If I say roll no 45 standup, or another way I will say Rahul standup, which one is better? Name i.e. Rahul Obviously.
  → If I say Roll no 45, may be at some point of time, Rahul whose roll no is 45, even rahul thinks who is 45 and if I directly say rahul then he will standup.

→ Rather than calling a student by Rollno, calling by his/her name would be better.

→ Same thing in our program if we use name in a program rather than numbers then it will increase the readability of the program.

→ It is easy to maintain that program.

→ If you have written a program, some other person is reading your program, if you have used names in that program, rather than numbers, for that person it will be easy to understand your program.

→ It is not compulsory, without "enum" also we can write our program. (increases readability & maintainence)

→ and if you want to modify something that will be easy if you use enum.

How we define enum-?

Syntax:

```
struct student
  {
    --
    ---   } list of members.
    --
  };
struct student S1;
void main()
  { struct student S2;

    ---
    ---
    ---
  }
```

enum defining:

```
enum days
{       Sunday;
        monday;
        tuesday;
        wednesday;
        thursday;
        friday;
        saturday;
};
// this is how we define enum.
   (our own datatype)
```

let us define enum in a proper way:

```
enum week_days {sun, mon, tue, wed, thur, fri, sat};
     ↓
    name
```

↳ we are listing the values here.
(meaning of this thing is how

You can declare a variable of `enum week_days` datatype.
(we have defined our own datatype.

But when we declare a variable of "enum weekdays" datatype, only the possible ways this variable can have is only 7 values here (sun --- saturday).

suppose if i declare `int a;` ⇒ so the possible values, a can hold is all the integer values.

`Char c;` ⇒ all the possible character values, 'c' can take.

How we can declare the above enum week_days datatype?

```
enum weekdays { sun, mon, tue, wed, thu, fri, sat }; ⇒ definition.
enum weekdays today, today1; ⇒ declaration.
```

weekdays → datatype.

today, today1 → this is enum variable and can have values from the list of values declared in enum weekdays datatype only.

We can assign something like below:

```
today = Tue;
today1 = sun;
```

These today, today1 variables can have values as only sun, mon, tue, wed, thu, fri and sat only (from this 7 values only, that's it.

NO other values are allowed.

Suppose i define the values till fri only.

```
enum weekdays { sun, mon, tue, wed, thu, fri };
```

if you write today = sat; ⇒ this would be wrong ↓ it is not allowed because it is not there in list.

If you want a program to be written like your variable can have only a set of values, at that point of time we use enum.

Ex: months ⇒ only 12 can be there.
weekday ⇒ only 7 can be there
directions ⇒ only 4 can be there

either one value, the variable can have (the variable you declare of that type).

At one time, the variable can have only a single value.

like `today = sun;` ⇒ Assigned ~~Value~~.

So If this situation arise that, a list of values for a specific set of values your variable should have, at that time you can use enum user-defined data type.

Second thing is:

Automatically, the compiler will initialize these 7 values with integers. like 0, 1, 2, 3, 4, 5, 6, 7.

```
enum weekdays { sun, mon, Tue, wed, thu, fri, sat };
                0   1    2    3    4    5    6
```

Suppose now if i print the value of today:

printf("%d", today); ⇒ it will print '0' because in today we have sun (by default).

Automatically the compiler will assign numeric values (integers Constants to these values (values of enum).

✓ Imp:

But if you assign your own values also. Like this

enum week days { sun = 1, mon, Tue, wed, thu, fri, sat };
2 3 4 5 6 7

we have assigned sun = 1

These next values will be assigned by "previous value + 1", automatically by the compiler.

But if you assign your numbers in any order, that is also fine.

enum weekdays { sun = 1, mon = 5, Tue, wed, thu, fri, sat };

this is also fine

But automatically tuesday will be assigned with "previous value + 1"

so tuesday will be assigned '6' by the compiler automatically.

printf("%d", today); it will print 6.

In any order you can assign the values.

If you assign sun = 1 and mon = 0;

enum weekdays { sun = 1, mon = 0, Tue, wed, thu, fri, sat };
1 2 3 4 5 6 7
0

sun = 1 and even tue = 1, this is also ok.

So the next value would be the previous value + 1.

We can assign the values according to yourself (according to the requirement.
numbers)

Why we use enum over macro's ?

Macro's also do the same thing, assign's name to an constant.

1. one difference between enum and macro : Scope.

Macro's are having global scope only.

#define ⇒ these are like preprocessor directives.

So it comes under pre-processing phase.

Whatever the lines which are prefixed with "#", those will come in preprocessor phase.

Suppose: `#define MAX 50` ⇒ so whenever you use MAX in program, that would be replaced with 50.

So this is having global scope, no local scope.

No local scope is allowed in macro's.

Wrong ⇒ even you use this line within any function, then also it is considered as a global scope not local to that function.

Wrong.

But enum can have both local as well as global scope.

Within a function also you can define & declare enum!
(Local scope).

```
void fun()
{
    enum colors {red, blue, green};
}
```

this type we can use within this function only
we cannot use it outside.

globally also you can declare.

(2) Automatically the compiler will assign values to enum variables.
But when you use macros, you have to explicitly give the value.

(3) → At one point of time, in enum we can take list of values.
→ But in enum, we cannot take these many, only one value we can take, if you have to take another value, you have to define another macro here.

The above are some plus points of using enum over macros

Next point about enum:

a) Within the same scope, i can use like below:

```
enum weekdays {sun, mon, tue, wed, thu, fri};  ✓
enum weekdays {sat};  ✓
enum weekdays {sat, sun};  ⇒ X.
```

this is wrong

Within the same scope, either in mains i am using these enum, enum, the duplicate values are not allowed of these enum.

But if the above enum i am using in another function, in that case we can write {sat, sun}; i.e. one enum in one function other enum in another function, allowed.

But in the same scope not allowed.

b) only Integral values are allowed in enum.

ex:

enum point {x=0, y=2.5};
this is not allowed

only integer values are allowed.

enum point {x=0, y=0}; ✓ is allowed.

c) When a variable should have set of possible values, then we can use enum.

here set of possible values are only week days, 7.

d) In switch-case also we can use enum.

How can we use enum in switch case?

When we are using case, in case we are using integers like case 0, case 1 ... so on (or) we can use characters 'a' like this is more readable than 0, 1, ...

But if you use enum, you can use case sun; → case : mon ... so on and in that case you can print like today is sunday, monday ... so on.

Here in case sun; ↳ we can use these words when we are using enum.

Suppose I have to write a program like:

If an Customer is female, she/he will get 50% discount.
If male - 25% discount.
If others - 60% discount.

So if we are using enum, then directly we can use:

```
if(customer == female)
{
    discount = 50%;
}
```

when we use enum only, then only we can write down like above, it is more understandable. rather than writing 0, 1 or something.

```
enum gender { male, female, others };
            0     1       2
enum gender g;
g = male;
if (g == male
    { discount=25%;
    }.
```

like this. you can write programs.

Assignment to students:

using switch case and enum - write a program.

programs on enum:

```
#include<stdio.h>
enum languages { c, cpp, java };
main()
{ enum languages var;
  printf("%d", sizeof(var));
  return 0;
}
```

Whenever you declare a variable of enum type, and at one time, it can take only one value from a set of values.

⇒ and by default the values of enum are considered as 0, 1, 2.... so on.

size of int on 64 bit machine is 4 bytes.
so size of var will be 4 bytes here.

② #include <stdio.h>
enum xenum { c, cpp, java };
enum yenum { c, cpp, java, xenum };

```
main()
{
    enum yenum var;
    printf("%d", sizeof(var));
    return 0;
}
```

Olp-)

one property we have discussed that in the same scope, same values of enum x enum and yenum are not allowed. but i.e. c, cpp, java are not allowed (duplicate).

but in enum yenum { i am using xenum }; ⇒ this is allowed, this will not give error.

enum within enum is allowed, but duplicates are not allowed.

Try: enum xenum = { c, cpp, java }; } Assignment: check out these
enum yenum = { xenum }; } and check what olp you will get -)

③ typedef enum {

male ;
female ;
} gender ;

void main()
{
gender var = female ;
printf ("%d", var) ; ← o/p = or error ?
return 0 ;
}

because I am not using enum gender var here ?

if getting 1 , then why it is not giving error ?

Designated Initialization of Structures (Additional topic) :

struct question
{ int a ;
float b ;
int c ;
} ;
void main()
{ struct question q = {.b = 1, .c = 10, .a = 18} ; → this is what is designated Initialization.
printf ("%d %f %d", q.a, q.b, q.c) ;
}

(.dot operator)

This is what is an example of designated Initialization of a structure.

Why are we calling this as so ?

Explanation:

Here we have a structure question, and 3 members are there, 2 are integer and one is float.

and in main() ⇒ We have initialized the values and printing those values.

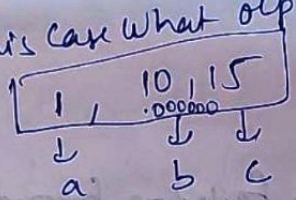
→ To access the members of structure we use the name of the variable (structure variable) dot operator (member of structure)

⇒ If you initialize like below rather than the initialization how we did in mains function of the program.

```
Struct question q = {1, 10, 15};
```

↓

In this case what o/p we will get ⇒?



⇒ If you initialized like this.

```
Struct question q = {.b=1, c=10, a=18};
```

dot operator is very important and

We cannot access the members of structure without dot operator.

.b=1 ⇒ means it is called as designated initialization.

I want this '1' to be given to this b.

" " " 10 " " " " c

I want this 18 to be given to this a

In a=18, b=1.00, c=10
0000

C programming Examples:

programs on enum:

a) #include <stdio.h>

enum year { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug
Sep, Oct, Nov, Dec };

int main()
{
int i;
for (i=Jan; i<=Dec; i++)
printf("%d", i);
return 0;
}

o/p:

0 1 2 3 4 5 6 7 8 9 10 11.

b) #include <stdio.h>

enum state { working=1, failed=0, freezed=0 };

Both will have value is 0's.

int main()
{ printf("%d, %d, %d", working, failed, freezed);
return 0;
}

o/p:

1, 0, 0.

c). We do not explicitly assign values to enum names, the compiler by default assigns values starting from 0.

In the following example, Sunday gets 0, Monday gets 1 ... so on.

```
#include <stdio.h>
enum day { sunday, monday, tuesday, wednesday,
           thursday, friday, saturday };
int main()
{
    enum day d = thursday;
    printf("the day no stored in d is %d", d);
    return 0;
}
```

Op:

The day no stored in d is 4.

d) we can assign values to some name in any order, All unassigned names get value as value of previous name + 1.

```
#include <stdio.h>
enum day { sunday=1, monday, tuesday=5, wednesday,
           thursday=10, friday, saturday };
int main()
{
    printf("%d %d %d %d %d %d %d",
    sunday, monday, tuesday, wednesday, thursday,
        friday, saturday);
    return 0;
}
```

Op: 1 2 5 6 10 11 12

e).

```
enum state { working, failed };
enum result { failed, passed };
int main()
{
return 0;
}
```

Olp: Compile Error: 'failed' has a previous declaration.

programs on Bitfields in C:

a) The following program is declaration of date without the use of bitfields.

```
#include<stdio.h>
struct date
{
    unsigned int d;
    unsigned int m;
    unsigned int y;
};
int main()
{
    printf("Size of date is %d bytes\n",
        sizeof(struct date));
    struct date dt = {01, 04, 2022};
    printf("Date is %d-%d-%d", dt.d, dt.m, dt.y);
}
```

Olp:

Size of date is 12 bytes

Date is :- 01/04/2022.

The above representation of 'date' takes 12 bytes on a compiler where an unsigned int takes 4 bytes.

Since we know that value of d (date) is always from 1 to 31.
Value of m (month) " " " 1 to 12.

We can optimize the space using bit fields.

If the same code is written using signed int and the value of the fields goes beyond the bits allocated to the variable and something interesting can happen.

program considering the same code but with signed integers:

```
#include<stdio.h>
// space optimized representation of the date.
struct date {
  // d has value between 0 and 31, so 5 bits are
  //sufficient in binary.

  int d:5;
// m has value between 0 and 15, so 4 bits are
//sufficient

  int m:4;
  int y;
};
int main()
{ printf("size of date is %lu bytes\n",
          sizeof(struct date));
  struct date dt = {31, 12, 2014};
  printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
  return 0; }
```

o/p: sized date is 8 bytes
Date is -1/-4

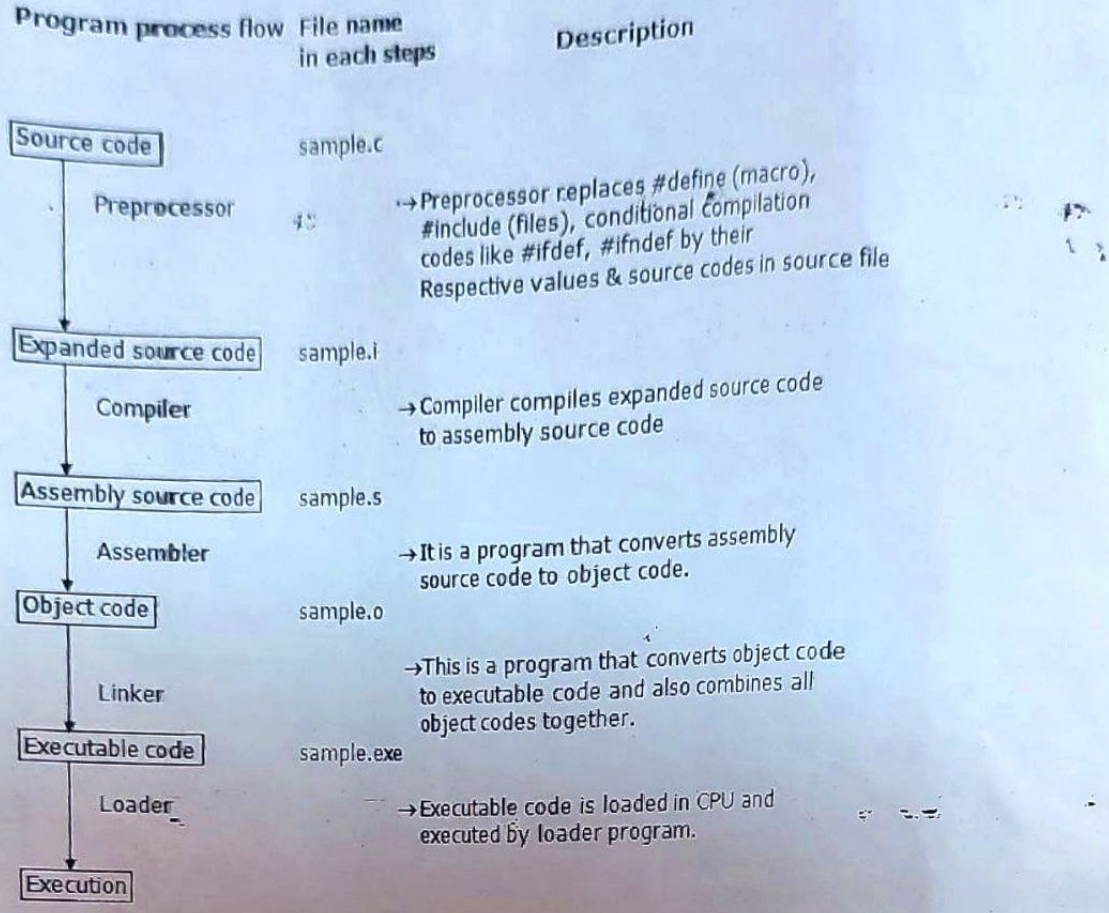
C PREPROCESSOR DIRECTIVES:

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with "#" symbol.

Below is the list of preprocessor directives that C programming language offers.

Preprocessor	Syntax/Description
Macro	**Syntax:** #define This macro defines constant value and can be any of the basic data types.
Header file inclusion	**Syntax:** #include <file_name> The source code of the file "file_name" is included in the main program at the specified place.
Conditional compilation	**Syntax:** #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.
Other directives	**Syntax:** #undef, #pragma #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program.

A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.



There are 4 regions of memory which are created by a compiled C program. They are,

1. **First region** – This is the memory region which holds the executable code of the program.
2. **2nd region** – In this memory region, global variables are stored.
3. **3rd region** – stack
4. **4th region** – heap

DO YOU KNOW DIFFERENCE BETWEEN STACK & HEAP MEMORY IN C LANGUAGE?

Stack	Heap
Stack is a memory region where "local variables", "return addresses of function calls" and "arguments to functions" are hold while C program is executed.	Heap is a memory region which is used by dynamic memory allocation functions at run time.
CPU's current state is saved in stack memory	Linked list is an example which uses heap memory.

DO YOU KNOW DIFFERENCE BETWEEN COMPILERS VS INTERPRETERS IN C LANGUAGE?

Compilers	Interpreters
Compiler reads the entire source code of the program and converts it into binary code. This process is called compilation.	
Binary code is also referred as machine code, executable, and object code.	Interpreter reads the program source code one line at a time and executing that line. This process is called interpretation.
Program speed is fast.	Program speed is slow.
One time execution. Example: C, C++	Interpretation occurs at every line of the program. Example: BASIC

KEY POINTS TO REMEMBER:

1. Source program is converted into executable code through different processes like precompilation, compilation, assembling and linking.
2. Local variables uses stack memory.
3. Dynamic memory allocation functions use the heap memory.

EXAMPLE PROGRAM FOR #DEFINE, #INCLUDE PREPROCESSORS IN C LANGUAGE:

- #define – This macro defines constant value and can be any of the basic data types.
- #include <file_name> – The source code of the file "file_name" is included in the main C program where "#include <file_name>" is mentioned.

```
#include <stdio.h>
  #define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'
  void main()
{
   printf("value of height    : %d \n", height );
   printf("value of number : %f \n", number );
```

```
    printf("value of letter : %c \n", letter );
    printf("value of letter_sequence : %s \n", letter_sequence);
    printf("value of backslash_char : %c \n", backslash_char);

}
```

OUTPUT:

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

EXAMPLE PROGRAM FOR CONDITIONAL COMPILATION DIRECTIVES:

A) EXAMPLE PROGRAM FOR #IFDEF, #ELSE AND #ENDIF IN C:

"#ifdef" directive checks whether particular macro is defined or not. If it is defined, "If" clause statements are included in source file.
Otherwise, "else" clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100

int main()
{
  #ifdef RAJU
  printf("RAJU is defined. So, this line will be added in " \
      "this C file\n");
  #else
  printf("RAJU is not defined\n");
  #endif
  return 0;
}
```

OUTPUT:

```
RAJU is defined. So, this line will be added in this C file
```

B) EXAMPLE PROGRAM FOR #IFNDEF AND #ENDIF IN C:

- #ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, "If" clause statements are included in source file.
- Otherwise, else clause statements are included in source file for compilation and execution.

```
#include <stdio.h>

#define RAJU 100
```

```
int main()
{
  #ifndef SELVA
  {
    printf("SELVA is not defined. So, now we are going to " \
        "define here\n");
    #define SELVA 300
  }
  #else
  printf("SELVA is already defined in the program");
  #endif
  return 0;
}
```

OUTPUT:

```
SELVA is not defined. So, now we are going to define here
```

C) EXAMPLE PROGRAM FOR #IF, #ELSE AND #ENDIF IN C:

- "If" clause statement is included in source file if given condition is true.
- Otherwise, else clause statement is included in source file for compilation and execution.

```
#include <stdio.h>
#define a 100
int main()
{
  #if (a==100)
  printf("This line will be added in this C file since " \
      "a \= 100\n");
  #else
  printf("This line will be added in this C file since " \
      "a is not equal to 100\n");
  #endif
  return 0;
}
```

OUTPUT:

```
This line will be added in this C file since a = 100
```

EXAMPLE PROGRAM FOR UNDEF IN C LANGUAGE:

This directive undefines existing macro in the program.

```
#include <stdio.h>
#define height 100
void main()
{
  printf("First defined value for height  : %d\n",height);
  #undef height          // undefining variable
  #define height 600     // redefining the same for new value
  printf("value of height after undef \& redefine:%d",height);
}
```

OUTPUT:

```
First defined value for height : 100
value of height after undef & redefine : 600
```

EXAMPLE PROGRAM FOR PRAGMA IN C LANGUAGE:

Pragma is used to call a function before and after main function in a C program.

```
#include <stdio.h>
  void function1( );
void function2( );
  #pragma startup function1
#pragma exit function2
  int main( )
{
  printf ( "\n Now we are in main function" ) ;
  return 0;
}

void function1( )
{
  printf("\nFunction1 is called before main function call");
}

void function2( )
{
  printf ( "\nFunction2 is called just before end of " \
        "main function" ) ;"
}
```

OUTPUT:

```
Function1 is called before main function call
Now we are in main function
Function2 is called just before end of main function
```

MORE ON PRAGMA DIRECTIVE IN C LANGUAGE:

Pragma command	Description
#Pragma startup <function_name_1>	This directive executes function named "function_name_1" before
#Pragma exit <function_name_2>	This directive executes function named "function_name_2" just before termination of the program.
#pragma warn – rvl	If function doesn't return a value, then warnings are suppressed by this directive while compiling.
#pragma warn – par	If function doesn't use passed function parameter , then warnings are suppressed
#pragma warn – rch	If a non reachable code is written inside a program, such warnings are suppressed by this directive.