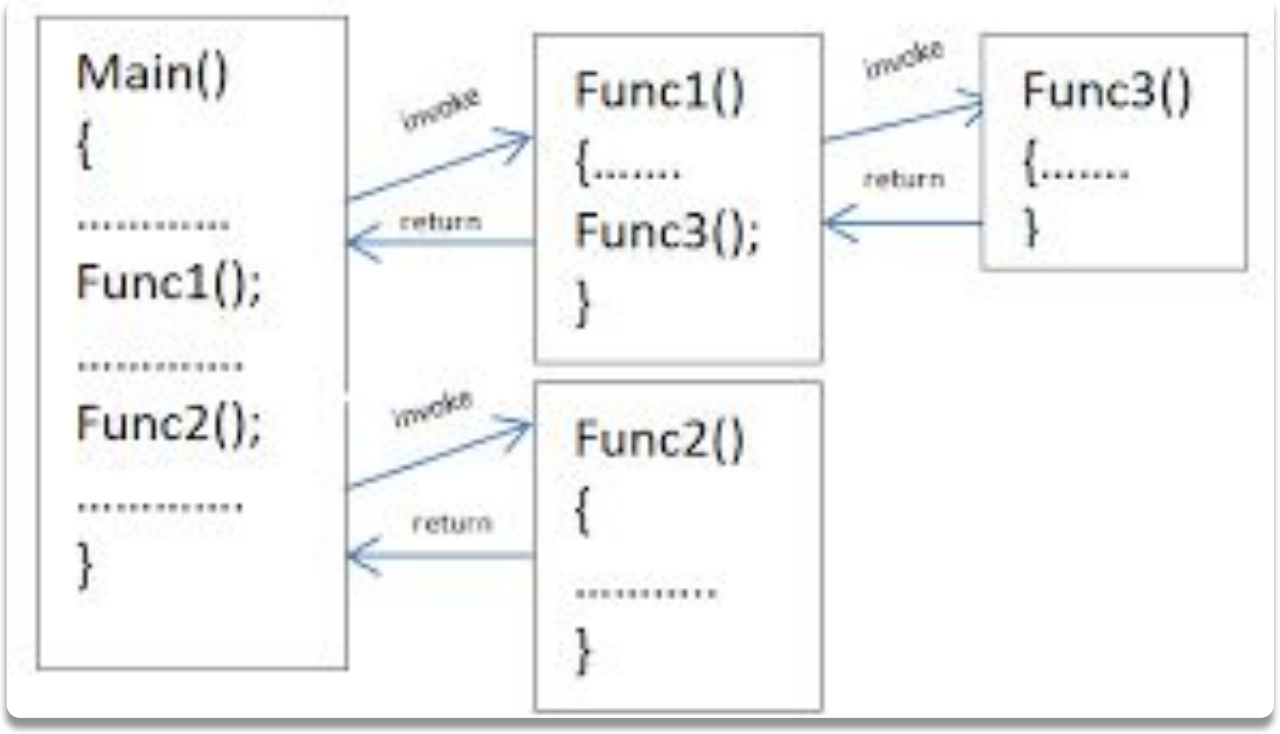
MODULE 4:

C Functions, Structure, Union:

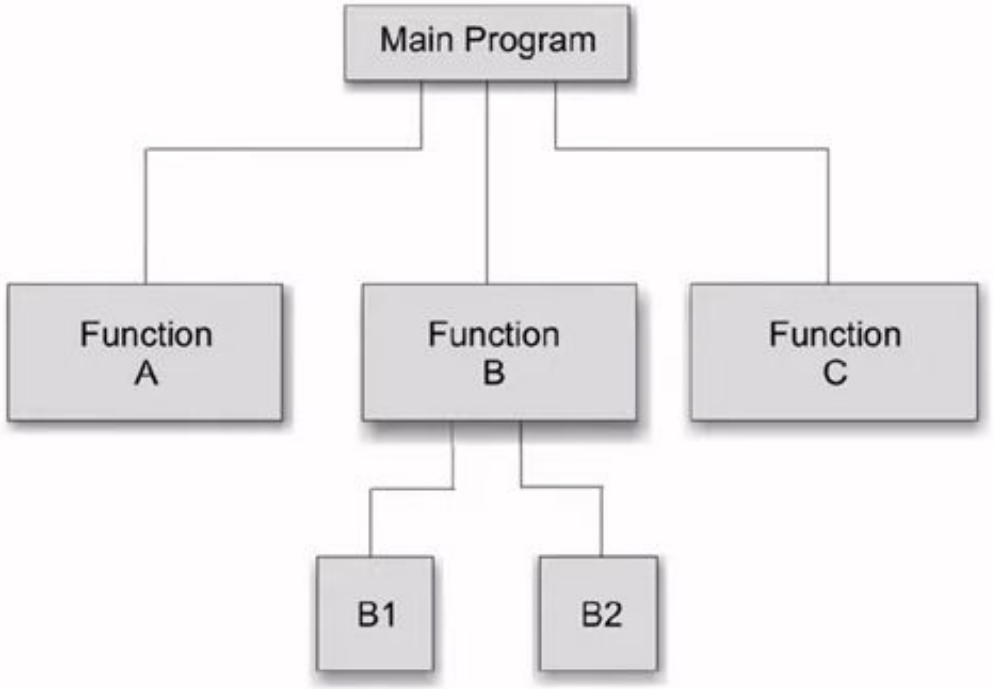
MOHITHA THOMAS
Asst Prof, EEE Dept
9847821112
mohitha@mace.ac.in
Room No: E 49

Modular programming

- It is defined as organizing a large program into small, independent program segments called **modules** that are separately named and individually callable program units.



Modular Programming



- A function is a block of code that performs a specific task.
- **Types of function**

There are two types of function in C programming:

- **Standard library functions**
- **User-defined functions**

Standard library functions

Standard library functions

- The standard library functions are built-in functions in C programming.
- These functions are defined in header files. For example,
- The **printf()** is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the **stdio.h** header file.
- Hence, to use the printf()function, we need to include the stdio.h header file using #include <stdio.h>.
- The **sqrt()** function calculates the square root of a number. The function is defined in the **math.h** header file.

User-defined function

- You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

USES OF C FUNCTIONS:

- C functions are used to avoid rewriting same logic/code again and again in a program.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, **re-usability** dividing a big task into small pieces to achieve **the functionality and to improve understandability** of very large C programs.

How user-defined function works?

```
#include <stdio.h>
void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

- The execution of a C program **begins** from the **main()** function.
- When the compiler encounters functionName();, control of the program jumps to

  void functionName()
- And, the compiler starts executing the codes inside functionName().
- The control of the program jumps back to the main() function once code inside the function definition is executed.

C FUNCTION DECLARATION, FUNCTION CALL AND FUNCTION DEFINITION

There are 3 aspects in each C function. They are,

1. **Function declaration or prototype**
2. **Function Call**
3. **Function definition**

Function declaration or prototype

- This informs compiler about the function name, function parameters and return value's data type.

Syntax of function prototype

- **returnType functionName**(type1 argument1, type2 argument2, ...);

Function call

Control of the program is transferred to the user-defined function by calling it.

- This calls the actual function

Syntax of function call

- functionName(argument1, argument2, ...);

Function definition

- This contains all the statements to be executed.

Syntax of function definition

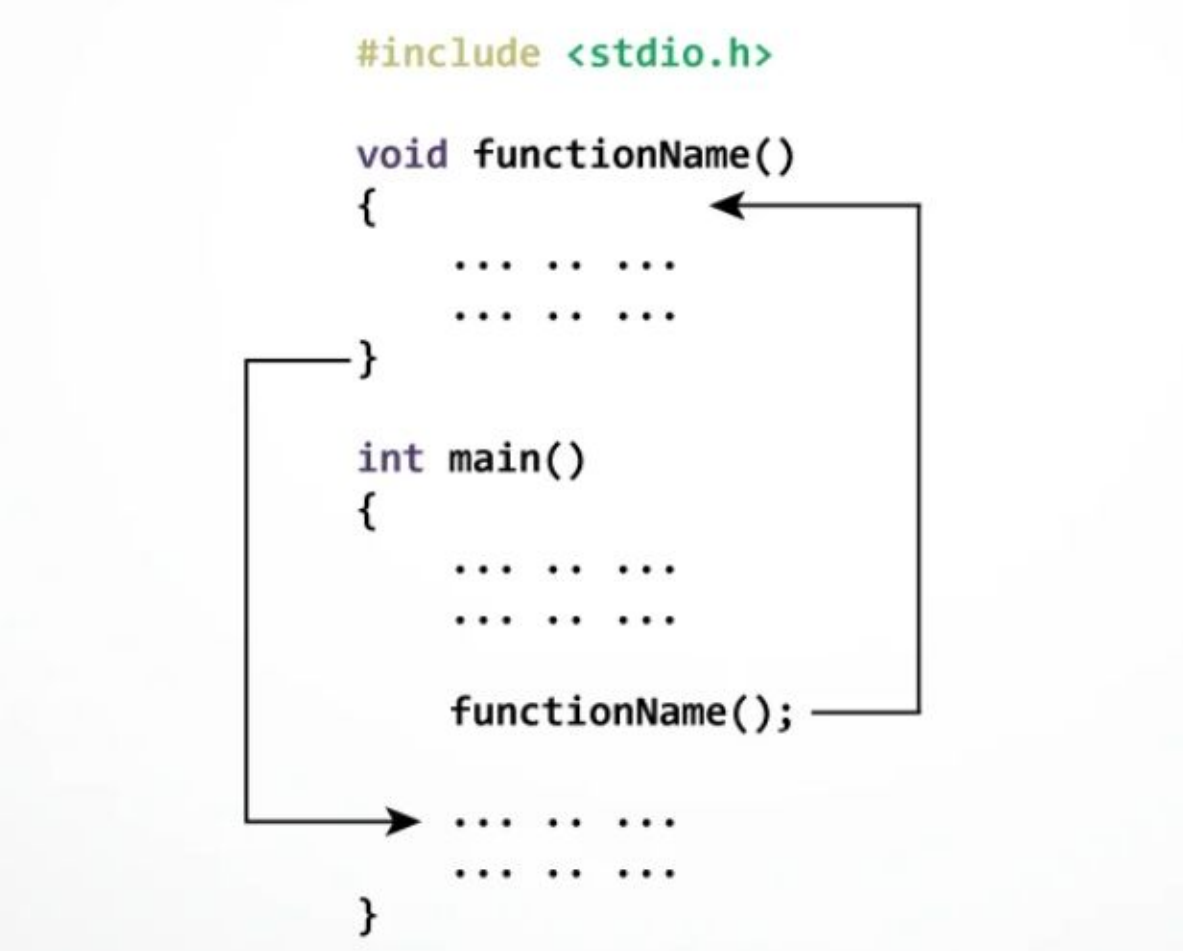
- returnType functionName(type1 argument1, type2 argument2, ...)

```
{
    //body of the function
}
```

Function definition

- A function definition in C programming consists of a **function header and a function body.**
- **Return Type:** A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- **Function Name**: The actual name of the function.
- **Arguments**: When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument.
- **Function Body**: The function body contains a collection of statements that define what the function does.

How function works in C?



```c
#include <stdio.h>
int addNumbers(int a, int b);          // function prototype

int main()
{
    int n1,n2,sum;

    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);         // function call
    printf("sum = %d",sum);

    return 0;
}

int addNumbers(int a, int b)          // function definition
{
    int result;
    result = a+b;
    return result;                   // return statement
}
```

Sum of two no.s using fn

MOHITHA\ADDFN.C

```
#include<stdio.h>
#include <conio.h>
int sum(int x, int y); //Fn Declaration
void main()
 {
 int a, b,add;
 clrscr();
 printf("Enter two numbers\n");
 scanf("%d%d",&a,&b);//a & b are called actual parametersss_
 add= sum(a,b); //fn call
 printf("The sum=%d",add);
 getch();
 }
 int sum(int x,int y) //fn definition , x & y are called formal/dummy p/m
        {
        int s;
        s=x+y;
        return(s);
        }
```

Formal and Actual Parameters

- There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case A is called the **"caller function"** and B is called the **"called function or callee function"**. Also, the arguments which A sends to B are called **actual arguments** and the parameters of B are called **formal arguments.**
- Formal Parameter: A variable and its type as they appear in the **prototype of the** function or method.
- Actual Parameter: The variable or expression defined in the main program.

Call by Value & Call by reference (Pass by value & Pass by reference)

S. No.	Call by Value	Call by Reference
1.	A copy of actual parameters is passed into formal parameters.	Reference of actual parameters is passed into formal parameters.
2.	Changes in formal parameters will not result in changes in actual parameters.	Changes in formal parameters will result in changes in actual parameters.
3.	Separate memory location is allocated for actual and formal parameters.	Same memory location is allocated for actual and formal parameters.

Call by value	Call by reference
Operation is done on the parameters.	Instead of passing values, address(ref)is passed to the p/m. fn **operates on addresses** rather than values

```
MOHITHA\ADDFN.C
S#include<stdio.h>
#include <conio.h>
int sum(int x, int y); //Fn Declaration
void main()
 {
 int a, b,add;
 clrscr();
 printf("Enter two numbers\n");
 scanf("%d%d",&a,&b);//a & b are called actual parameters
 add= sum(a,b); //fn call
 printf("The sum=%d",add);
 getch();
 }
 int sum(int x,int y) //fn definition , x & y are called formal/dummy p/m
        {
        int s;
        s=x+y;
        return(s);
        }
```

```
S//with arguments w/o return//
#include<stdio.h>
#include <conio.h>
void sum(int x, int y); //Fn Declaration
void main()
 {
 int a, b,add;
 clrscr();
 printf("Enter two numbers\n");
 scanf("%d%d",&a,&b);//a & b are called actual parameters
 sum(a,b); //fn call
 getch();
 }
void sum(int x,int y) //fn definition , x & y are called formal/dummy p/m
        {
        int s=0;
        s=x+y;
        printf("The sum=%d",s);
        }
```

```c
S//with arguments w/o return//
#include<stdio.h>
#include <conio.h>
void sum(int x, int y); //Fn Declaration
void main()
 {
 int a, b,add;
 clrscr();
 printf("Enter two numbers\n");
 scanf("%d%d",&a,&b);//a & b are called actual parameters
 sum(a,b); //fn call
 getch();
 }
void sum(int x,int y) //fn definition , x & y are called formal/dummy p/m
        {
        int s=0;
        s=x+y;
        printf("The sum=%d",s);
        }
```

```
S_ //w/o arguments w/0 return//
#include<stdio.h>
#include <conio.h>
void sum(); //Fn Declaration
void main()
{
 sum(); //fn call
 getch();
 }
void sum() //fn definition , x & y are called formal/dummy p/m
        {
        int s=0;
        int a,b;
        clrscr();
        printf("Enter two numbers\n");
        scanf("%d%d",&a,&b);//a & b are called actual parameters
        s=a+b;
        printf("The sum=%d",s);
        }
```


MOHITHA\FN4.C
#include <conio.h>
void power(int , int);
void main()
{
int x,n;
clrscr();
printf("Enter the number & power\n");
scanf("%d%d",&x,&n);
power(x,n);
getch();
}
void power(int a,int b)
{
int p=1,i;
for(i=0;i<b;i++)
{
p=p*a;
}
printf("power=%d",p);
}

Factorial of a number using fn

```
void fact(int); //Fn Declaration
void main()
 {
 int x;
 clrscr();
 printf("Enter the number\n");
 scanf("%d",&x);
 fact(x); //fn call
 getch();
 }
void fact(int a)
      {
        int f=1,i;
        if(a==0)
                {
                printf("factorial is %d",1);
                }
```

```
        else
        {
                for(i=1;i<=a;i++)
                  {
                  f=f*i;
                  }
                printf("factorial=%d",f);
        }
    }
```

Fact using Recursive fn

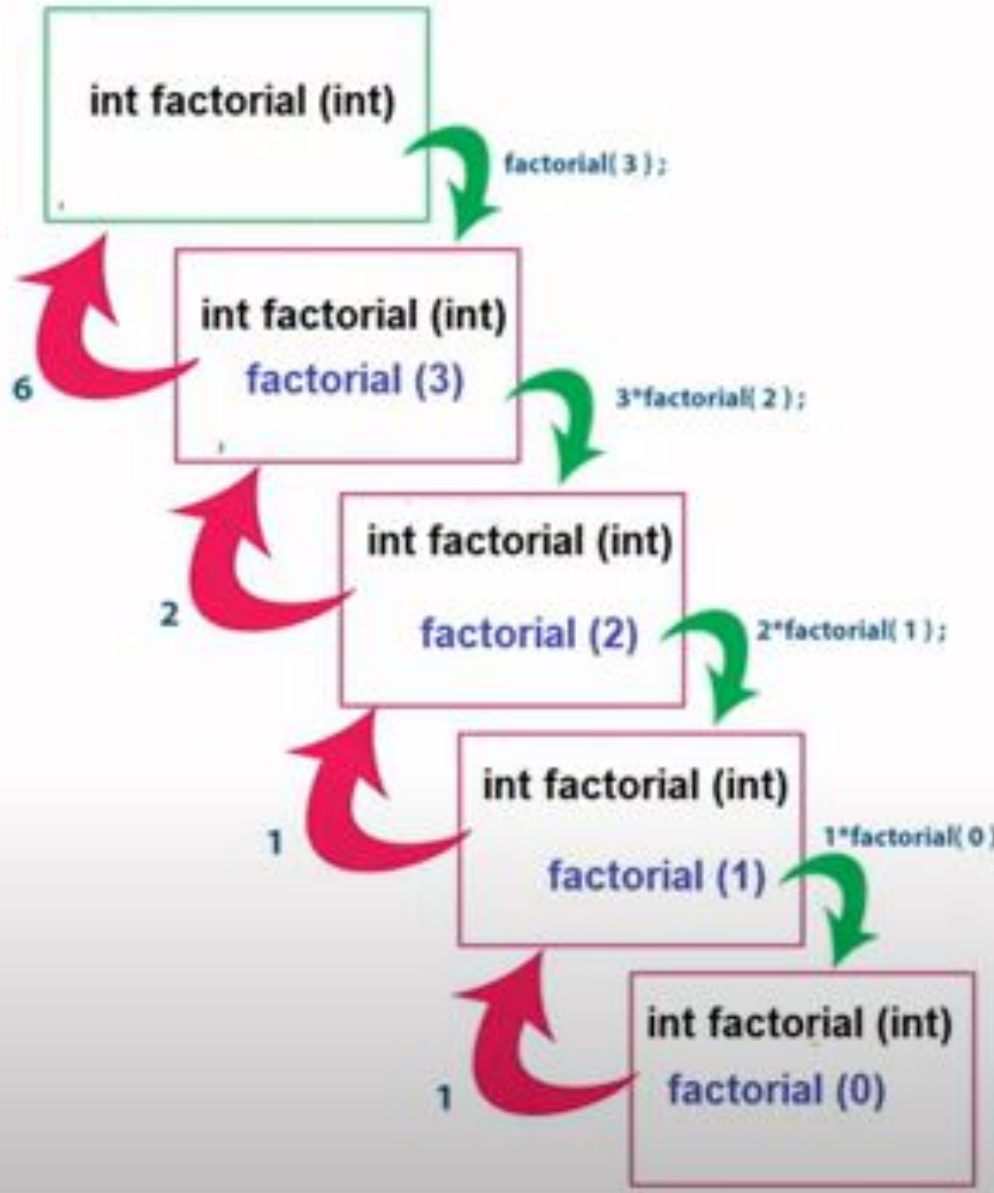
MOHITHA\FACTFN.C

MOHITHA\RECRSIVE.C

```c
#include<stdio.h>
#include<conio.h>
int fact(int a); //Fn Declaration
void main()
 {
 int x,f;
 clrscr();
 printf("Enter the number\n");
 scanf("%d",&x);
 f=fact(x); //fn call
 printf("facorial=%d",f);
 getch();
 }
int fact(int n)
      {
        int factorial;
        if(n==0)
         {
         return(1);
         }
        else
         {
        factorial=n*fact(n-1);
        return(factorial);
         }
      }
```

When function *factorial (3)* is called

- The **factorial()** function call is initiated from main() function with the value 3.
- Inside the factorial() function, the function calls factorial(2), factorial(1) and factorial(0) are called recursively.
- In this program execution process, the function call factorial(3) remains under execution till the execution of function calls factorial(2), factorial(1) and factorial(0) gets completed.
- Similarly the function call factorial(2) remains under execution till the execution of function calls factorial(1) and factorial(0) gets completed.
- In the same way the function call factorial(1) remains under execution till the execution of function call factorial(0) gets completed.



Passing1 D array to function_
Pgm to find largest element in an array

```c
#include<stdio.h>
#include<conio.h>
int largest(int a[],int size); //Fn Declaration
void main()
 {
 int i,n,large, arr[50];
 clrscr();
 printf("Enter no. of elements in the array\n");
 scanf("%d",&n);
 printf(" Enter the elements of the array\n");
 for(i=0;i<n;i++)
  {
   scanf("%d",&arr[i]);
  }
 large=largest(arr,n); //fn call
 printf("largest=%d",large);
 getch();
 }
```

```c
int largest(int a[], int size)
{
int l=a[0],i;
for(i=0;i<size;i++)
{
if(a[i]>l)
l=a[i];
}
return(l);
}
```

Pgm to find sum of array elements

MOHITHA\ARRAYSUM.C

[■] MOHITHA\FNSUMARR.C

```c
#include<stdio.h>
#include<conio.h>
int sum(int a[],int size); //Fn Declaration
void main()
 {
 int i,n,sum1, arr[50];
 clrscr();
 printf("Enter no. of elements in the array\n");
 scanf("%d",&n);
 printf(" Enter the elements of the array\n");
 for(i=0;i<n;i++)
  {
   scanf("%d",&arr[i]);
  }
 sum1=sum(arr,n); //fn call
 printf("sum=%d",sum1);
 getch();
 }
```

```c
int sum(int a[], int size)
{
int sum2=0,i;
for(i=0;i<size;i++)
{
sum2=sum2+a[i];
}
return(sum2);
}
```

String to fn_ Palindrome

MOHITHA\FN_PALIN.C

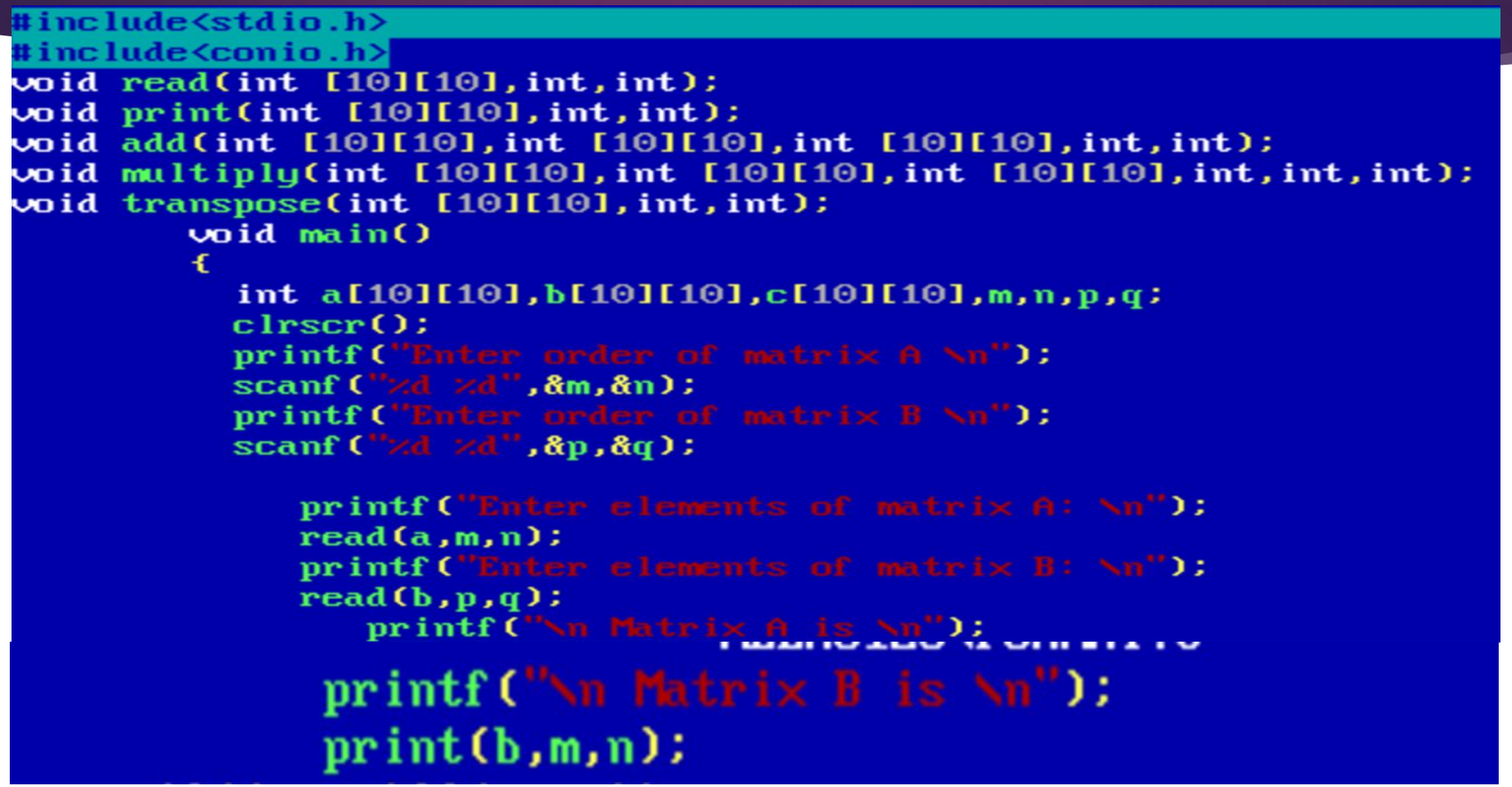
```c
#include<stdio.h>
#include<conio.h>
#include<string.h>
int compare(char s[],int length); //Fn Declaration
void main()
 {
 char s[50];
 int l,p;
 clrscr();
 printf("Enter a string\n");
 gets(s);
 l=strlen(s);
 p=compare(s,l);
 if(p==0)
        printf("String is palindrome\n");
 else
        printf("String is not a palindrome\n");
 getch();
 }
```

```c
int compare(char s[],int length)
{
int flag=0,i;
for(i=0;i<length/2;i++)
 {
  if((s[i]!=s[length-i-1]))
   {
    flag++;
    break;
   }
 }
return(flag);
}
```

Passing 2D array to fn

- In the fn definition , we must indicate that the array has two dimensions by **including two sets of brackets.**
- **The size of second dimension must be specified.**

Eg: int add(int a**[ ][10],** int m, int n);



```
#include<stdio.h>
#include<conio.h>
void read(int [10][10],int,int);
void print(int [10][10],int,int);
void add(int [10][10],int [10][10],int [10][10],int,int);
void multiply(int [10][10],int [10][10],int [10][10],int,int,int);
void transpose(int [10][10],int,int);
        void main()
        {
          int a[10][10],b[10][10],c[10][10],m,n,p,q;
          clrscr();
          printf("Enter order of matrix A \n");
          scanf("%d %d",&m,&n);
          printf("Enter order of matrix B \n");
          scanf("%d %d",&p,&q);

             printf("Enter elements of matrix A: \n");
             read(a,m,n);
             printf("Enter elements of matrix B: \n");
             read(b,p,q);
                printf("\n Matrix A is \n");
                printf("\n Matrix B is \n");
                print(b,m,n);
```


if((m==p)&&(n==q))
{
add(a,b,c,m,n);
printf("\n Sum Matrix is \n");
print(c,p,q);
}
if(n!=p)
printf("\n Matrix Multiplication is not possible \n");
else
{
multiply(a,b,c,m,n,q);
printf("\n Product Matrix is \n");
print(c,m,q);
}
if(m==n)
{
printf(" \n Transpose of Matrix A is \n");
}
getch();
}

```
void read(int a[10][10],int m,int n)
{
    int i,j;
    for(i=0;i<m;i++)
    {
      for(j=0;j<n;j++)
      {
        scanf("%d", &a[i][j]);
      }
    }
 }
    void print(int a[10][10],int m,int n)
    {
      int i,j;
      for(i=0;i<m;i++)
      {
         printf("\n");
         for(j=0;j<n;j++)
          {
            printf("%d\t", a[i][j]);
          }
```

```
          }
       }
void add(int a[10][10],int b[10][10],int c[10][10],int p,int q)
{
   int i,j;
     for(i=0;i<p;i++)
     {
       for(j=0;j<q;j++)
       {
         c[i][j] = a[i][j] + b[i][j];
       }
     }
  }
```

```c
void transpose(int a[10][10],int m,int n)
{
  int i,j;
  for(i=0;i<m;i++)
  {
    printf("\n");
    for(j=0;j<n;j++)
    {
          printf("%d\t", a[j][i]);
    }
  }
}
```

```c
void multiply(int a[10][10],int b[10][10],int c[10][10],int q,int m,int n)
{
 int i,j,k;
 for(i=0; i<m; i++)
 {
   for(j=0; j<q; j++)
   {
      c[i][j]=0;
      for(k=0; k<n; k++)
      {
         c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
    }
  }
}
```

Matrix addition using fn

Structure

- used for defining a collection of variables under a **single name**. Structures help programmers to group elements of **different data types** into a single logical unit. Unlike an array, a structure can contain **many different data types (int, float, char, etc.).**

Array Vs Structure

Array	Structure
Stores a set of data elements of the same data type in contiguous memory locations	Stores different data types as a single unit
It is possible to access an array element using the index	It is possible to access a property of a structure using the structure name and the dot operator
No keyword	"struct" keyword used
Each element has the same size	Size of the elements can be different
Requires less time to access	Requires more time to access

Structure Definition

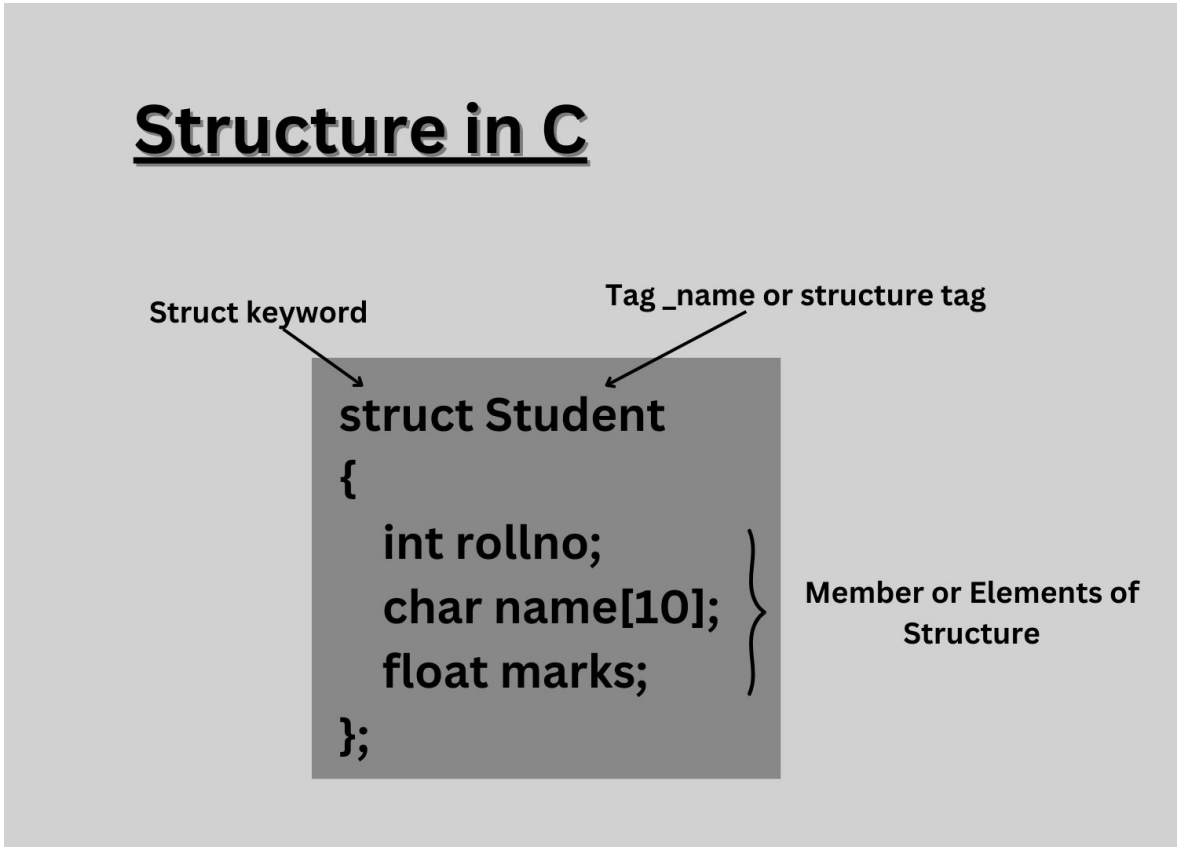
The general format of a structure definition is as follows

```
struct structure_name

{
        data_type       member1;
        data_type       member2;
        ---------------------------
        ---------------------------

};
```

Defining Structure



Structure Variable Declaration

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

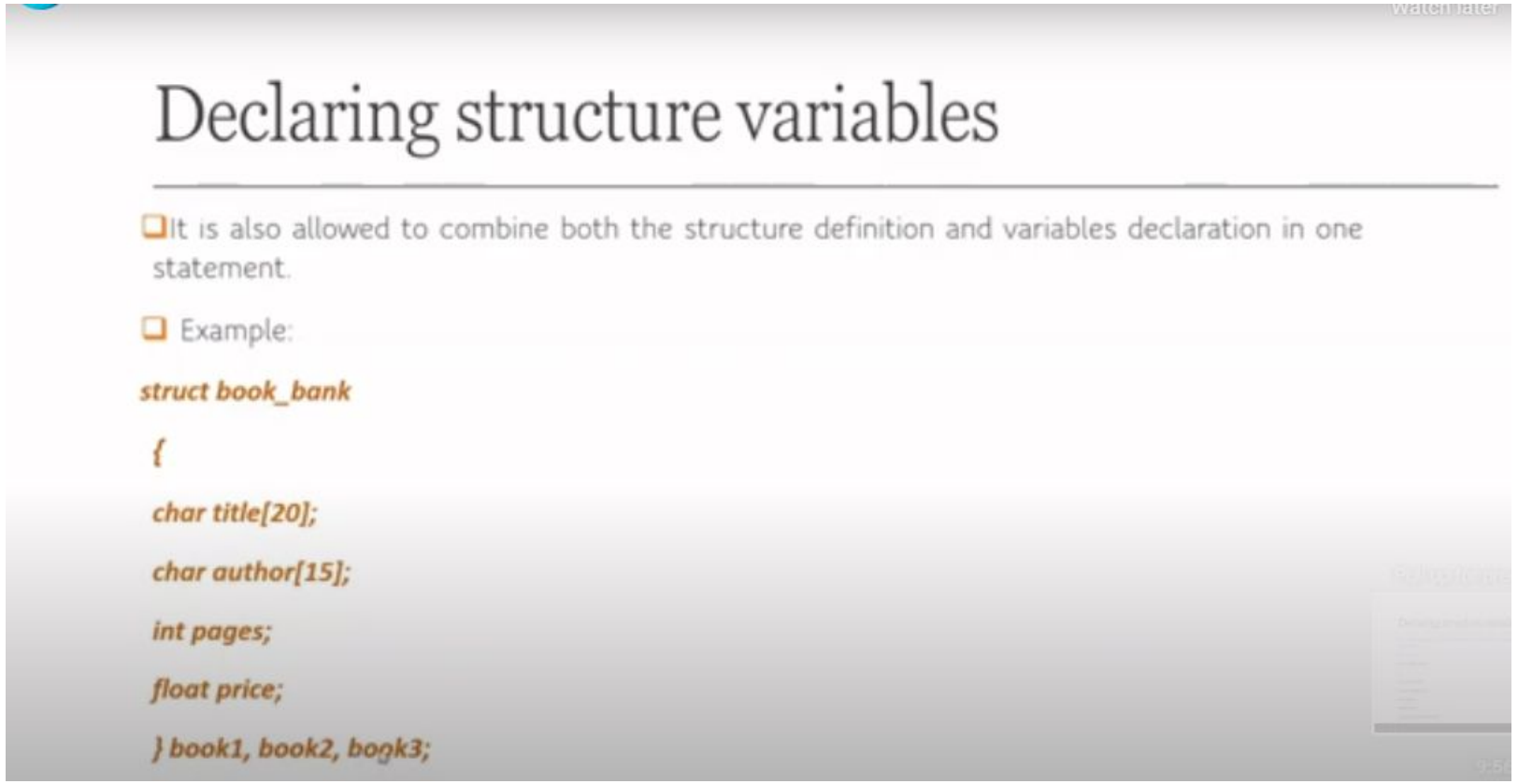
```
struct student
{
 int rno;
 char name[20];
 float marks;
};
struct student a;
```

Accessing Structure member

Accessing Structure Members in C:

1. Array elements are accessed using the Subscript variable, Similarly Structure members are accessed using dot [.] operator.
2. (.) is called as "Structure member Operator".
3. Use this Operator in between **"Structure name"** & **"member name"**

Declaring structure variable –method 1



Declaring structure variable –method 2

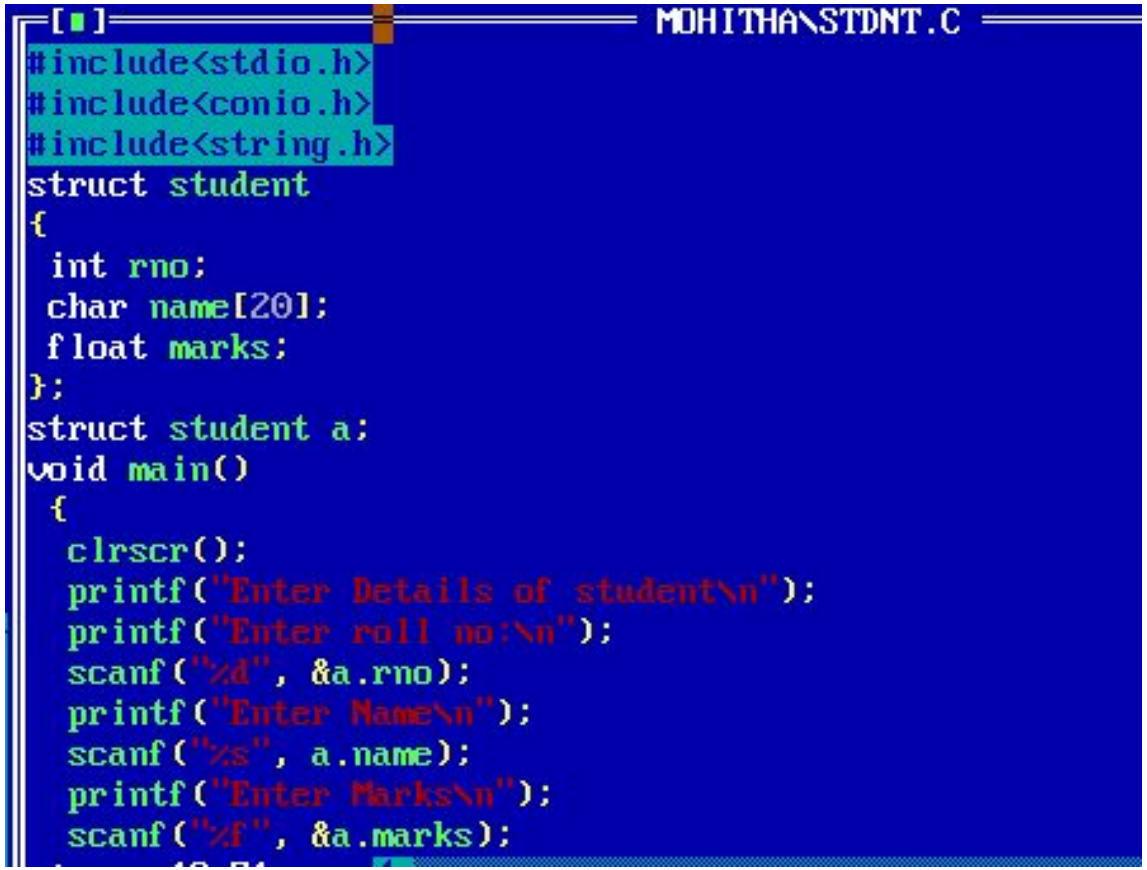
Declaring structure variables

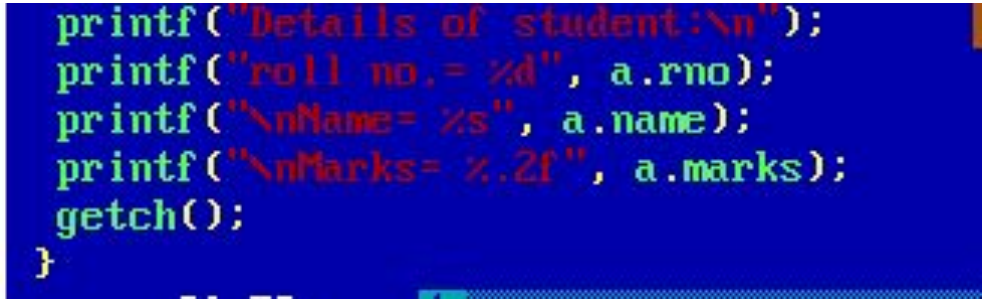
- For example,

struct book_bank book1, book2, book3;

declares book1, book2, and book3 as variables of type struct book_bank.

- Remember that the members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as book1.
- When the compiler comes across a declaration statement, it reserves memory space for the structure variables.


[■] MOHITHA\STDNT.C
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct student
{
int rno;
char name[20];
float marks;
};
struct student a;
void main()
{
clrscr();
printf("Enter Details of student\n");
printf("Enter roll no:\n");
scanf("%d", &a.rno);
printf("Enter Name\n");
scanf("%s", a.name);
printf("Enter Marks\n");
scanf("%f", &a.marks);


printf("Details of student:\n");
printf("roll no.= %d", a.rno);
printf("\nName= %s", a.name);
printf("\nMarks= %.2f", a.marks);
getch();
}

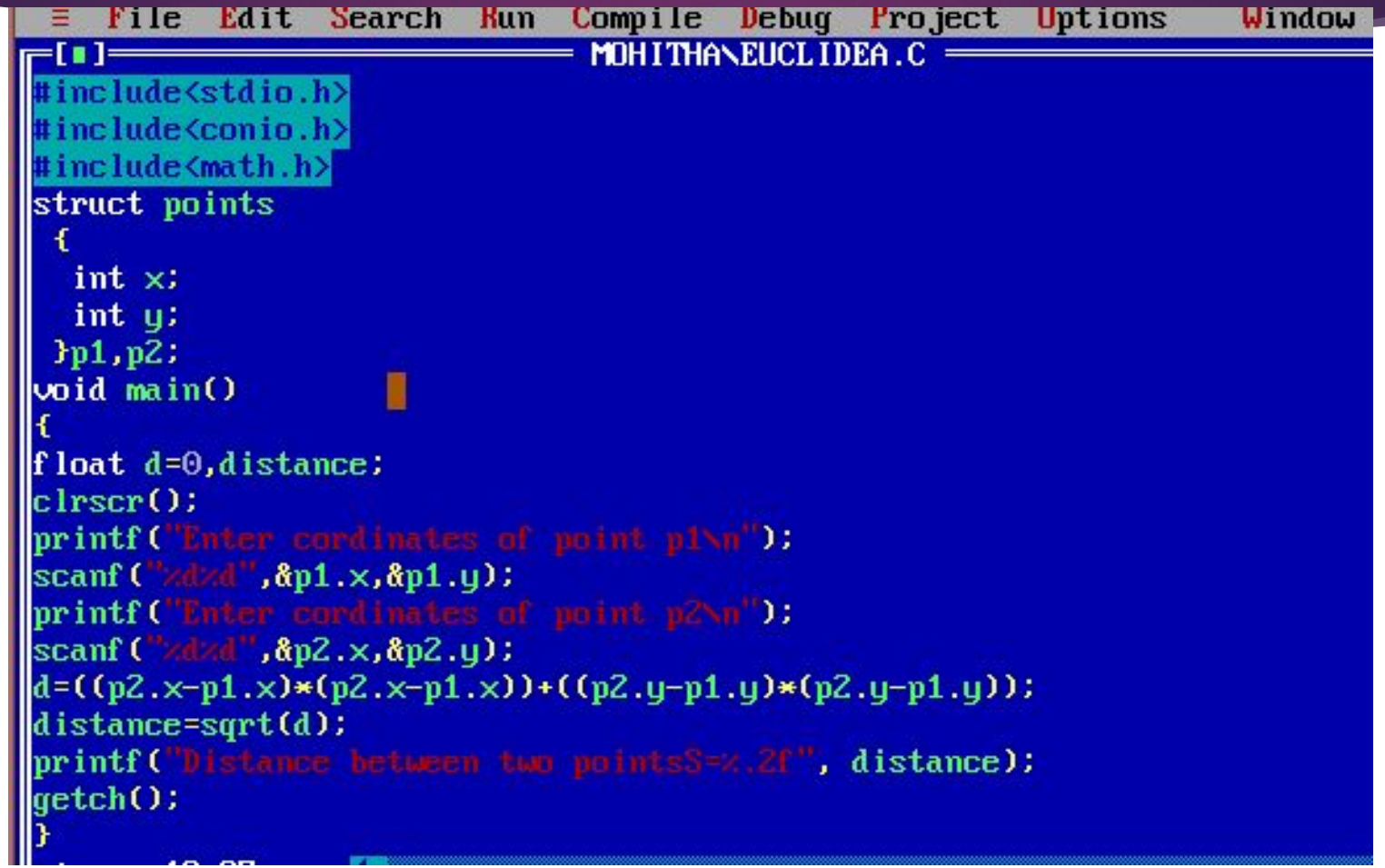
14. Using structure, read and print data of n employees (*Name, Employee Id, and Salary*).

```
MOHITHA\EMPLO
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct employee
{
 char name[80];
 int salary;
 int employeeid;
};
struct employee e[100];

void main()
{
int n, i;
  clrscr();
  printf("Enter no. of employees\n");
  scanf("%d", &n);
for(i=0;i<n;i++)
 {
  printf("\nEnter Name\n");
  scanf("%s",e[i].name);
  printf("\nEnter salary\n");
  scanf("%d",&e[i].salary);
  printf("\nEnter employee id:\n");
  scanf("%d", &e[i].employeeid);
 }

  for(i=0;i<n;i++)
    {
     printf("Employee %d details",i+1);
     printf("\nEmployee name: %s\n", e[i].name);
     printf("\nEmployee salary:%d\n",e[i].salary);
     printf("\nEmployee id:%d\n",e[i].employeeid);
    }
 getch();
}
```

13. Read two input each representing the distances between two points in the Euclidean space, store these in structure variables, and add the two distance values.



Union

STRUCTURE

- Structure is a heterogeneous collection data items.
- struct keyword is used to create a structure.
- **Syntax of structure:**

```
struct structure name
  {
    member definition;
    member definition;
    ...
    member definition;
  };
```

UNION

- Union is also heterogeneous collection of data items. But union allocates the memory size of the largest element and that memory is shared by other elements.
- union keyword is used to create a union.
- **Syntax of Union:**

```
union [union name]
  {
    member definition;
    member definition;
    ...
    member definition;
  };
```

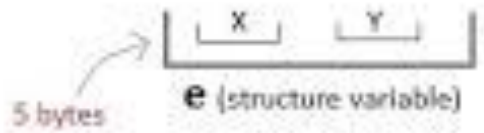
Structure Vs Union

union	structure
union keyword is used to define the union type.	structure keyword is used to define the union type.
Memory is allocated as per largest member.	Memory is allocated for each member.
All field share the same memory allocation.	Each member have the own independent memory.

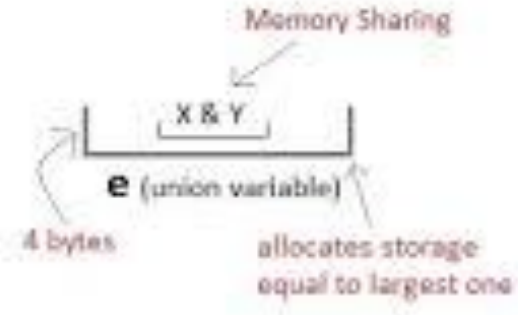
We can access only one field at a time.
 	
union Data {
int a; // can't use both a and b at once
char b;
} Data;
 	
union Data x;
x.a = 3; // OK
x.b = 'c'; // NO! this affects the value of x.a
	We can any member any time.
	
	struct Data {
	int a; // can use both a and b simultaneously
	char b;
	} Data;
	
	struct Data y;
	y.a = 3; // OK
	y.b = 'c'; // OK

Altering the value of the member affect the value of the other member.	Altering the value of the member does not affect the value of the other member.

```
struct Emp
{
 char X ;     // size 1 byte
 float Y ;    // size 4 byte
} e ;
```




```
union Emp
{
 char X ;
 float Y ;
} e ;
```



15. Declare a union containing 5 string variables (*Name, House Name, City Name, State, and Pin code*) each with a length of C_SIZE (user defined constant). Then, read and display the address of a person using a variable of the union.

File Edit Search Run Compile Debug Project Options Window He

MOHITHA\PERSUNI.C

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define C_SIZE 50
union person
{
 char name[C_SIZE];
 char hname[C_SIZE];
 char cityname[C_SIZE];
 char state[C_SIZE];
 char pincode[C_SIZE];
};
union person p;
void main()
{
  clrscr();
  printf("Enter name\n");
  scanf("%s", p.name);
  printf("Enter house name\n");
  scanf("%s", p.hname);
  printf("Enter city name\n");
  scanf("%s", p.cityname);
  printf("Enter state name\n");
  scanf("%s", p.state);
  printf("Enter pincode\n");
  scanf("%s", p.pincode);
printf("Details Entered is\n");
printf("Name: %s",p.name);
printf("\nHouse Name: %s",p.hname);
printf("\nCity Name: %s",p.cityname);
printf("\nState: %s",p.state);
printf("\nPincode: %s",p.pincode);
getch();
}
```

1:59

Thank you...