Modular Programming

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

There are many advantages to using Modular Software and Programming compared to other methods.

- **Development can be divided:** Allows development to be divided by splitting down a program into smaller programs in order to execute a variety of tasks.
- **Readable programs:** Helps develop programs that are much easier to read since they can be enabled as user-defined functions.
- **Programming errors are easy to detect:** Minimizes the risks of ending up with programming errors and also makes it easier to spot errors, if any.
- **Allows re-use of codes:** A program module is capable of being re-used in a program which minimizes the development of redundant codes
- **Improves manageability:** Having a program broken into smaller sub-programs allows for easier management.
- **Collaboration:** With Modular Programming, programmers can collaborate and work on the same application.

Function

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions

Types of Functions

There are two types of function in C programming:

- Standard library functions
- User-defined 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.
- The sqrt() function calculates the square root of a number. The function is defined in the math.h header file.
- The strlen() function calculates the length of a given string. The function is defined in the string.h header file.

User-defined function

User can also create functions as per their own needs. Such functions created by users are known as user-defined functions.

Advantages of user-defined function

- The program will be easier to understand, maintain and debug.
- Reusable codes that can be used in other programs
- A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

Three parts of a user defined functions are:

1) Function Declaration or Prototype
2) Function Definition
3) Function Call

Function Declaration

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body. A function prototype gives information to the compiler that the function may later be used in the program.

Syntax

return_type function_name(parameter list);

Function Definition

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function:

- **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.

Syntax:

```
return_type function_name( argument list ) {
    body of the function
}
```

Function Call

Control of the program is transferred to the user-defined function by calling it. When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

Syntax:

```
functionName(parameter list);
```

Example:

```
int add(int,int,int); // function declaration
…..
……
int add(int a, int b, int c)    // function definition
{
        int sum = a+ b + c;
        return sum;
}
```

```
void main()
{
        int x=10,y=20,z=30;
        int res = add(x,y,z); // function call
        print("Result=%d",res);
}
```

Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the above example, the value of the **sum** variable is returned to the main function. The **res** variable in the main() function is assigned this value.

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 corresponding to a formal parameter that appears in the function or method call in the calling environment.

In the above example, x,y and z in the main function are the actual parameter of add function. Formal parameter of add function are a, b and c.

Pass by Value

In this parameter passing technique, a copy of actual parameter is passed to formal parameter. As a result, any changes or modification happened to formal parameter won't reflect back to actual parameter. This can be explained with an example of swapping program.

```
#include<stdio.h>
int swap(int a,int b)
{
        int temp=a;
        a= b;
        b=temp;
}

void main()
{
        int x,y;
        printf("Enter the numbers:");
        scanf("%d%d",&x,&y);
        printf("Before swapping : x=%d\ty=%d\n",x,y);
        swap(x,y);
        printf("After swapping : x=%d\ty=%d",x,y);
}
```

Output
Enter the numbers:10 20
Before swapping: x=10 y=20
After swapping : x=10 y=20

In above program we expect the program to swap the value of x and y after calling the function swap with x and y as actual parameter. But swapping does not place as this uses call by value as parameter passing technique.

During Swap call, a copy of actual parameters are created and changes are made on that copy. So the value of x and y does not changes.

After reading the value of x and y

x: 10	y: 20

During swap call a copy is created and passed to formal parameter

x: 10	a: 10
y: 20	b: 20

After the swap call, formal parameter get swapped but actual parameter remains the same.

x: 10	a: 20
y: 20	b: 10

Types of user defined functions

1) **No arguments passed and no return value**

```
// function to read two number and print its sum
void add()
{
        int a,b,sum;
        printf("Enter the values of a & b");
        scanf("%d%d",&a,&b);
        sum = a+b;
        printf("Sum=%d",sum);
}
```

2) **No arguments passed but return value**

```
// function to read two number and return its sum

int add()
{
        int a,b,sum;
        printf("Enter the values of a & b");
        scanf("%d%d",&a,&b);
        sum = a+b;
        return sum;
}
```

3) **Arguments passed but no return value**

```
// function that takes two arguments and print their sum.
void add(int a,int b)
{
        int sum= a + b;
        printf("Sum=%d",sum);
}
```

4) **Arguments passed with return value**

```
// function that takes two arguments and print their sum.
```

```
int add(int a,int b)
{
        return a + b;
}
```

- **Write a program to perform arithmetic operations using function**

```
#include<stdio.h>
int add(int a,int b)
{
    return a+b;
}
int diff(int a,int b)
{
    return a-b;
}
int mul(int a,int b)
{
    return a*b;
}
float div(int a,int b)
{
    return (float)a/b;
}
int mod(int a,int b)
{
    return a%b;
}

void main()
{
    int x,y;
    printf("Enter the values:");
    scanf("%d%d",&x,&y);
    printf("Sum=%d\n",add(x,y));
    printf("Difference=%d\n",diff(x,y));
    printf("Multiply=%d\n",mul(x,y));
    printf("Division=%f\n",div(x,y));
    printf("Modulo=%d\n",mod(x,y));
}
```

Output
Enter the values:21 8

Sum=29
Difference=13
Multiply=168
Division=2.625000
Modulo=5

- **Write a program to display prime numbers upto a range using function.**

```
// function takes an argument n, if n is prime it will return 1 otherwise 0

#include<stdio.h>
int checkPrime(int n)
{
    int flag=0,i;
    for(i=2;i<=n/2;i++)
    {
            if(n%i == 0)
            {
                    flag =1;
                    break;
            }
    }
    if(flag == 0)
            return 1;
    else
            return 0;
}


void main()
{
    int n,i;
    printf("Enter the range:");
    scanf("%d",&n);
    for(i=2;i<=n;i++)
    {
            if(checkPrime(i) == 1)
                    printf("%d\t",i);
    }
}
```

Output
Enter the range: 25

2 3 5 7 11 13 17 19 23

Recursive Function

A function that calls itself is known as a recursive function. And, this technique is known as recursion. While using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop. Recursion makes program elegant. However, if performance is vital, use loops instead as recursion is usually much slower.

- **Write a program to find factorial of number using recursive function**

```
#include<stdio.h>
int fact(int n)
{
    if( n == 0)
            return 1;
    return n*fact(n-1);
}

void main()
{
    int n;
    printf("Enter the number:");
    scanf("%d",&n);
    printf("Factorial=%d",fact(n));
}
```

Output
Enter the number:5
Factorial=120

- **Write a program to find nCr and nPr.**

```
#include<stdio.h>
int fact(int n)
{
    if( n == 0)
            return 1;
    return n*fact(n-1);
}
```

```
void main()
{
    int n,r;
    float C,P;
    printf("Enter the numbers:");
    scanf("%d%d",&n,&r);
    P = (float) fact(n) / fact(n-r);
    C = (float)fact(n) / (fact(r) * fact(n-r));
    printf("nCr=%f\n",C);
    printf("nPr=%f",P);
}
```

Output

Enter the numbers:5 3

nCr=10.000000

nPr=60.000000

- Write a program to find of series 1 + 1/2! + 1/ 3! ..

```
#include<stdio.h>
int fact(int n)
{
    if( n == 0)
            return 1;
    return n*fact(n-1);
}

void main()
{
    int n,i;
    float sum;
    printf("Enter the limit:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
            sum = sum + 1.0 / fact(i);
    printf("Sum=%f",sum);
}
```

Output
Enter the limit:5
Sum=1.716667

- **Write a recursive function to print Fibonacci series**

```
#include<stdio.h>
int fib(int n)
{
    if(n == 1)
        return 0;
    else if(n == 2)
        return 1;
    return fib(n-2)+fib(n-1);
}

void main()
{
    int n,i;
    printf("Enter the limit:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
        printf("%d\t",fib(i));
}
```

Output
Enter the limit:7
0 1 1 2 3 5 8

- **Write a recursive program to print sum of digit of a number**

```
#include<stdio.h>
int sod(int n)
{
    if(n<=0)
        return 0;
    return n%10 + sod(n/10);
}

void main()
{
    int n;
    printf("Enter the number:");
    scanf("%d",&n);
    printf("Sum=%d",sod(n));
}
```

Output
Enter the number:124

Sum=7

- **Write a recursive function to find sum of first n natural numbers**

```
#include<stdio.h>
int sum(int n)
{
    if(n<=0)
            return 0;
    return n + sum(n-1);
}
void main()
{
    int n;
    printf("Enter the number:");
    scanf("%d",&n);
    printf("Sum=%d",sum(n));
}
```

Output
Enter the number:5
Sum=15

Passing an array as Parameter

Like the value of simple variables, it is also possible to pass the values of an array to a function. To pass a single dimensional array to a function, it is sufficient to list the name of the array without any subscripts and the size of the array as arguments

Rules to pass an Array to Function

- The function must be called by passing only the name of the array.
- In function definition, formal parameter must be an array type; the size of the array does not need to be specified.
- The function prototype must show that the argument is an array.

- **Write a function to sort an array.**

```
#include<stdio.h>
int sort(int A[],int n)
{
    int i,j,temp;
```

```
    for(i=0;i<n-1;i++)
    {
        for(j=0;j<n-i-1;j++)
        {
            if(A[j]>A[j+1])
            {
                temp  = A[j];
                A[j]  = A[j+1];
                A[j+1]= temp;
            }
        }
    }
}

void main()
{
    int A[30];
    int i,n;
    printf("Enter the limit:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter the element:");
        scanf("%d",&A[i]);
    }
    sort(A,n);
    printf("Sorted Array\n");
    for(i=0;i<n;i++)
        printf("%d\t",A[i]);
}
```

Output

```
Enter the limit:5
Enter the element:17
Enter the element:23
Enter the element:5
Enter the element:2
Enter the element:9
Sorted Array
2       5       9       17      23
```

- **Write a program to sort the matrix rowwise.**

```
#include<stdio.h>
int sort(int A[],int n)
{
    int i,j,temp;
    for(i=0;i<n-1;i++)
    {
        for(j=0;j<n-i-1;j++)
        {
            if(A[j]>A[j+1])
            {
                temp  = A[j];
                A[j]  = A[j+1];
                A[j+1]= temp;
            }
        }
    }
}

void main()
{
    int A[30][30];
    int i,n,m,j;
    printf("Enter the order of matrix:");
    scanf("%d%d",&m,&n);
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("Enter the element:");
            scanf("%d",&A[i][j]);
        }
    }
    for(i=0;i<m;i++)
    {
        sort(A[i],n);
        for(j=0;j<n;j++)
        {
            printf("%d\t",A[i][j]);
        }
        printf("\n");
    }
}
```

Output

```
Enter the order of matrix:2 4
Enter the element:12
Enter the element:4
Enter the element:25
Enter the element:7
Enter the element:8
Enter the element:5
Enter the element:16
Enter the element:2
4      7      12      25
2      5      8      16
```

Passing 2 D array as parameter

Note: While passing 2D as parameter we need to mention the max size of element of each row that is column.

- **Write a program to pass a 2 D matrix as parameter and find its sum of all the elements.**

```
#include<stdio.h>
// function that takes a 2 D and its order
int sum(int A[][30],int m,int n)
{
    int i,j,sum =0;
    for(i=0;i<m;i++)
    {
            for(j=0;j<n;j++)
            {
                    sum = sum + A[i][j];
            }
    }
    printf("Sum=%d",sum);
}

void main()
{
    int A[30][30];
    int i,n,m,j;
    printf("Enter the order of matrix:");
    scanf("%d%d",&m,&n);
    for(i=0;i<m;i++)
    {
```

```
            for(j=0;j<n;j++)
            {
                printf("Enter the element:");
                scanf("%d",&A[i][j]);
            }
        }
        sum(A,m,n);
    }
```

Output
Enter the order of matrix:2 2
Enter the element:1
Enter the element:2
Enter the element:3
Enter the element:4
Sum=10

SCOPE, VISIBILITY AND LIFETIME OF VARIABLE

In C, not only do all the variables have a data type, they also have a storage class. The following variable storage classes are most relevant to functions

- Automatic Variables
- External or Global Variables
- Static Variables
- Register Variables

Automatic Variables

Automatic variables are declared inside a function in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic. Automatic variables are therefore private or local to the function in which they are declared. Because of this property, automatic variables are also referred to as local or internal variables.

A variable declared inside a function without storage class specification is by default an automatic variable.

```
void main(){
    int num;
```

```
        ----------
        ----------
    }
```

is same as

```
    void main(){
        auto int num;
        ----------
        ----------
    }
```

Example

```
#include<stdio.h>

void func1()
{
    int  max=10;
    printf("Max in func1()=%d\n",max);
}
void func2(){
    int  max=20;
    printf("Max in func2()=%d\n",max);
}
void main(){
    int max=30;
    func1();
    func2();
    printf("Max in main()=%d\n",max);
}
```

Output

```
Max in func1()=10
Max in func2()=20
Max in main()=30
```

External Variables

Variables that are both alive and active throughout the entire program are known as external variables. They are called global variables. Unlike local variables, global variables can be accessed by any function in the program. External variables are declared outside a function

```
#include<stdio.h>

float pi=3.14; // One way of declaring external variable

float area(int r){
        return pi*r*r;
}
float perimeter(int r){
        return 2 * pi * r;
}
void main(){
        // extern float pi=3.14;  Another way of declaring external variable
        int r;
        float a,p;
        printf("Enter the radius:");
        scanf("%d",&r);
        a=area(r);
        p=perimeter(r);
        printf("Area=%f\n",a);
        printf("Perimeter=%f\n",p);
}
```

Output
Enter the radius:5
Area=78.500000
Perimeter=31.400002

More example to show the property of global variable.

```
#include<stdio.h>
int max; // global variable
void func1(){
        int  max=10; // local variable
        printf("Max in func1()=%d\n",max);
}
void func2(){
        max=20; // resets max value to 20
        printf("Max in func1()=%d\n",max);
}
void main(){
        max=40; //set max value to 40
        func1();
        func2();
```

```
        printf("Max in main()=%d\n",max);
}
```

```
Max in func1()=10
Max in func1()=20
Max in main()=20
```

Why the value of max in main() print as 20? [Hint: main uses global scope of max.]

Static Variables

As the name suggests, the value of static variables persists until the end of the program. A variable can be declared static using the keyword **static** like

static *int x;*

A static variable may be either an internal type or an external type depending on the place of declaration. Internal static variable are those which are declared inside the function. The score of internal static variable are similar to auto variables, except that they remain in existence throughout the remainder of the program. Therefore the internal static variables can be used to retain values between the function calls.

```
#include<stdio.h>

void func1(){
        static int  x=10; //static variable
        x++;
        printf("x in func1()=%d\n",x);
}
void func2(){
        int x=10; // local variable
        x++;
        printf("x in func2()=%d\n",x);
}
void main(){
        func1();
        func1();
        func2();
        func2();
}
```

Output
x in func1()=11
x in func1()=12
x in func2()=11
x in func2()=11

Register Variables

We can tell the compiler that a variable should be kept in one of the machine's registers instead of keeping in the memory. Since a register access is fast than a memory access, keep in the frequently accessed variables in the register will lead to faster execution of programs. This is done as follows

register int count;

Since only a few variables can be places in the register, it is important to carefully select the variables for these purposes. However C will automatically convert register variables into non register variable once limit is reached.

Storage Class	Where declared	Visibility	Lifetime
extern	Before all functions in a file (cannot be initialized) extern and the file where originally declared as global.	Entire file plus other files where variable is declared.	Global
Static	Before all the function in a file Or Inside a function	Only in that file OR Only in that function	Global
None or **auto**	Inside a function	Only in that function or block	Until end of function
Register	Inside a function or block	Only in that function or block	Until end of function

Structure

A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type. 'struct' keyword is used to create a structure. A structure variable can either be declared with structure declaration or as a separate

declaration like basic types. Consider we want to create the structure of a person with following variable name, age and address. Then such a structure can be created as

```
struct Person
{
        char name[30];
        int age;
        char addr[50];
}
```

The general format of a structure definition is as follows

```
struct structure_name{
        data_type       member1;
        data_type        member2;
        ---------------------
        ---------------------
};
```

In defining a structure you may note the following syntax:

- The template is terminated with a semicolon.
- While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.

Difference between structure and Array

Array	Structure
An array is a collection of related data elements of same type.	Structure can have elements of different types.
An array is derived data type	Structure is a programmer defined one
Any array behaves like built in data type. All we have to do is to declare an array variable and use it	Structure we have to design and declare a data structure before the variable of that type are declared and used.

Declaring structure variable

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of any other data type. It includes the following elements.

1. The keyword struct
2. The structure tag name (structure name)

3. List of variable names separated by commas.

4. A terminating semicolon.

Example: struct Person p1,p2,p3; // created structure variable for person Structure.

- **Write a program to create a structure employee with member variables name, age, bs, da, hra and tsalary. Total Salary is calculate by the equation tsalary= (1+da+hra)* bs. Read the values of an employee and display it.**

```
#include<stdio.h>
struct Employee{
    char name[30];
    int age;
    float bs;
    float da;
    float hra;
    float tsalary;
};
void main(){
    struct Employee e;

    printf("Enter the name:");
    scanf("%s",e.name);
    printf("Enter the age:");
    scanf("%d",&e.age);
    printf("Enter the basic salary:");
    scanf("%f",&e.bs);
    printf("Enter the da:");
    scanf("%f",&e.da);
    printf("Enter the hra:");
    scanf("%f",&e.hra);
    e.tsalary=(1+e.da+e.hra)*e.bs;

    printf("Name=%s\n",e.name);
    printf("Age=%d\n",e.age);
    printf("Basic Salary=%.2f\n",e.bs);
    printf("DA=%.2f\n",e.da);
    printf("HRA=%.2f\n",e.hra);
    printf("Total Salary=%.2f\n",e.tsalary);
}
```

Output

```
Enter the name:Sangeeth
Enter the age:31
Enter the basic salary:10000
Enter the da:12
Enter the hra:7.5
Name=Sangeeth
Age=31
Basic Salary=10000.00
DA=12.00
HRA=7.50
Total Salary=205000.00
```

- **Write a program to create a structure Complex with member variable real and img. Perform addition of two complex numbers using structure variables.**

```
#include<stdio.h>
struct Complex{
    int real;
    int img;
};

void main(){

    struct Complex a,b,c;
    printf("Enter the real and img part of a:");
    scanf("%d%d",&a.real,&a.img);
    printf("Enter the real and img part of b:");
    scanf("%d%d",&b.real,&b.img);
    c.real = a.real + b.real;
    c.img = a.img + b.img;
    printf("c = %d + %di\n",c.real,c.img);
}
```

Output

```
Enter the real and img part of a:10 20
Enter the real and img part of b:30 40
c = 40 + 60i
```

Array of Structures

- **Declare a structure namely Student to store the details (roll number, name, mark_for_C) of a student. Then, write a program in C to find the average mark obtained by the students in a class for the subject Programming in C (using the field mark_for_C). Use array of structures to store the required data.**

```
#include<stdio.h>
struct Student{
    char name[30];
    int rollnum;
    int mark_for_C;
};

void main(){

    struct Student s[30];
    int i,n,sum=0;
    float avg;
    printf("Enter the no of Student:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
            printf("Enter the Student name:");
            scanf("%s",s[i].name);
            printf("Enter the Student rollnum:");
            scanf("%d",&s[i].rollnum);
            printf("Enter the Student Mark for C:");
            scanf("%d",&s[i].mark_for_C);
    }
    printf("Name\tRoll Number\tMark for C\n");
    for(i=0;i<n;i++)
    {
            printf("%s\t%d\t%d\n",s[i].name,s[i].rollnum,s[i].mark_for_C);
            sum = sum + s[i].mark_for_C;
    }
    avg = sum / (float)n;
    printf("Average Mark=%.2f\n",avg);
}
```

Output
Enter the no of Student:3
Enter the Student name:Sangeeth

```
Enter the Student rollnum:27
Enter the Student Mark for C:35
Enter the Student name:Pratheesh
Enter the Student rollnum:24
Enter the Student Mark for C:40
Enter the Student name:Poornima
Enter the Student rollnum:26
Enter the Student Mark for C:45
Name    Roll Number     Mark for C
Sangeeth        27      35
Pratheesh       24      40
Poornima        26      45
Average Mark=40.00
```

- **Write a program to create a structure employee with member variables name, age, bs, da, hra and tsalary. Total Salary is calculate by the equation tsalary= (1+da+hra)* bs. Read the values of 3 employees and display details based descending order of tsalary.**

```
#include<stdio.h>
struct Employee{
    char name[30];
    int age;
    float bs;
    float da;
    float hra;
    float tsalary;
};
void sort(struct Employee e[],int n)
{
    int i,j;
    struct Employee t;
    for(i=0;i<n-1;i++)
    {
            for(j=0;j<n-i-1;j++)
            {
                    if(e[j].tsalary < e[j+1].tsalary)
                    {
                            t = e[j];
                            e[j]=e[j+1];
                            e[j+1]=t;
                    }
            }
    }
```

```
}
void main(){
    struct Employee e[5];
    int i;
    for(i=0;i<3;i++)
    {
            printf("Enter the name:");
            scanf("%s",e[i].name);
            printf("Enter the age:");
            scanf("%d",&e[i].age);
            printf("Enter the basic salary:");
            scanf("%f",&e[i].bs);
            printf("Enter the da:");
            scanf("%f",&e[i].da);
            printf("Enter the hra:");
            scanf("%f",&e[i].hra);
            e[i].tsalary=(1+e[i].da+e[i].hra)*e[i].bs;
    }
    sort(e,3);
    printf("Name\t    Age\tBasic Salary\tDA    \t HRA    \t Total Salary\n");
    for(i=0;i<3;i++)
    {
            printf("%s\t%d\t%.2f\t",e[i].name,e[i].age,e[i].bs);
            printf("%.2f\t%.2f\t%.2f\n",e[i].da,e[i].hra,e[i].tsalary);
    }

}
```

Output

```
Enter the name:Sangeeth
Enter the age:31
Enter the basic salary:14000
Enter the da:6
Enter the hra:7.5
Enter the name:Poornima
Enter the age:28
Enter the basic salary:15000
Enter the da:7.6
Enter the hra:8
Enter the name:Pratheesh
Enter the age:29
Enter the basic salary:15000
Enter the da:8
```

Enter the hra:9

Name	Age	Basic Salary	DA	HRA	Total Salary
Pratheesh	29	15000.00	8.00	9.00	270000.00
Poornima	28	15000.00	7.60	8.00	249000.00
Sangeeth	31	14000.00	6.00	7.50	203000.00

Union

A union is a user-defined type similar to struct in C programming. We use the union keyword to define unions. When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Example of Employee Union creation and declartion

```
union Employee
{
    char name[30];
    int age;
    double salary;
}

union Employee e;
```

Difference between structure and union

Structure	Union
struct keyword is used to define a structure	**union** keyword is used to define a union
Members do not share memory in a structure.	Members share the memory space in a union
Any member can be retrieved at any time in a structure	Only one member can be accessed at a time in a union.
Several members of a structure can be initialized at once.	Only the first member can be initialized.
Size of the structure is equal to the sum of size of the each member.	Size of the union is equal to the size of the largest member.

Predict the output

```
#include<stdio.h>
struct Person
{
    char pincode[6]; // Size = 6 bytes
    int age;  // Size = 4 bytes
    double salary;// Size = 8 bytes
};
union Employee
{
    char pincode[6];
    int age;
    double salary;
};
void main()
{
    struct Person p;
    union Employee e;
    printf("Size of Structure Person=%d\n",sizeof(p));
    printf("Size of Union Employee=%d",sizeof(e));
}
```

Output

Size of Structure Person=24
Size of Union Employee=8