GXEST204

PROGRAMMING IN C

MODULE 3

Functions - Function definition, Function call, Function prototype, Parameter passing; Recursion; Passing array to function; Macros – Defining and calling macros; Command line Arguments. Structures - Defining a Structure variable, Accessing members, Array of structures, Passing structure to function; Union. Storage Class - Storage Classes associated with variables: automatic, static, external and register.

FUNCTIONS

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. We can divide up our code into separate functions. Logically the division is such that each function performs a specific task.

In 'C' programming, functions are divided into two types:

1. Library functions
2. User-defined functions

- The difference between the library and user-defined functions in C is that we do not need to write a code for a library function. It is already present inside the header file which we always include at the beginning of a program. You just have to type the name of a function and use it along with the proper syntax. printf, scanf are the examples of a library function.
- Whereas, a user-defined function is a type of function in which we have to write a body of a function and call the function whenever we require the function to perform some operation in our program.
- A user-defined function in C is always written by the user.
- C programming functions are divided into three activities in case of user defined functions such as,
  - Function declaration(Function prototype)
  - Function definition

- Function call

Function Declaration

Function declaration means writing a name of a program. It is a compulsory part for using functions in code. In a function declaration, we just specify the name of a function that we are going to use in our program like a variable declaration. We cannot use a function unless it is declared in a program. A function declaration is also called "Function **prototype**."

return_data_type function_name (data_type arguments);

- The **return_data_type**: is the data type of the value function returned back to the calling statement.
- The **function_name**: is followed by parentheses
- **Arguments** names with their data type declarations optionally are placed inside the parentheses.

Function Definition

Function definition means just writing the body of a function. A body of a function consists of statements which are going to perform a specific task. A function body consists of a single or a block of statements. It is also a mandatory part of a function.

```
return_datatype function name(datatype arg1, ..)
{
        Body of the function
}
```

Function call

A function call means calling a function whenever it is required in a program. Whenever we call a function, it performs an operation for which it was designed. A function call is an optional part of a program. 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.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

To call a function, write the function's name followed by two parentheses `()` and a semicolon `;`

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function – call by value and call by reference.

Call by value

In call by value method of parameter passing, the values of actual parameters are copied to the function's formal parameters.

- There are two copies of parameters stored in different memory locations.
- One is the original copy and the other is the function copy.
- Any changes made inside functions are not reflected in the actual parameters of the caller.

```
// C program to illustrate call by value
#include <stdio.h>
// Function Prototype
void swapx(int x, int y);
int main()
{
   Int a = 10, b = 20;
     // Pass by Values
   swapx(a, b);
   printf("In the Caller:\na = %d b = %d\n", a, b);
   return 0;
}
  // Swap functions that swapstwo values
void swapx(int x, int y)
{
   int t;
  t = x;
   x = y;
   y = t;
  printf("Inside Function:\nx = %d y = %d\n", x, y);
}
```

Call by reference

In call by reference method of parameter passing, the address of the actual parameters is passed to the function as the formal parameters.

- Both the actual and formal parameters refer to the same locations.
- Any changes made inside the function are actually reflected in the actual parameters of the caller.

```
/ C program to illustrate Call by Reference
#include <stdio.h>
// Function Prototype
void swapx(int*, int*);
int main()
{
    int a = 10, b = 20;
   // Pass reference
    swapx(&a, &b);
    printf("Inside the Caller:\na = %d b = %d\n", a, b);
   return0;
}
  // Function to swap two variablesby references
void swapx(int* x, int* y)
{
    intt;
    t = *x;
    *x = *y;
    *y = t;

    printf("Inside the Function:\nx = %d y = %d\n", *x, *y);
}
```

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

Function with no argument and no return value

```
#include<stdio.h>
void my_function()
{
  printf("This is a function that takes no argument, and returns nothing.");
}
main()
{
```

```
  my_function();
}
```

Function with no argument and with return value

```
#include<stdio.h>
int my_function()
{
  printf("This function takes no argument, But returns 50");
  return50;
}
main()
{
  int x;
  x = my_function();
  printf("Returned Value: %d", x);
}
```

Function with argument and no return value

```
#include<stdio.h>
void my_function(int x)
{
  printf("This function is taking %d as argument, but returns nothing", x);
  return50;
}
main()
{
  int x;
  x =10;
  my_function(x);
}
```

Function with argument and with return value

```
#include<stdio.h>
int my_function(int x)
{
```

```
  printf("This will take an argument, and will return its squared value
");
  return x * x;
}
```

```
main()
{
  int x, res;
  x =12;
  res = my_function(12);
  printf("Returned Value: %d", res);
}
```

RECURSION

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function. Recursion is often termed as 'Circular Definition'.

```
void recursion()
{
        recursion(); /* function calls itself */
}
int main()
{
        recursion();
}
```

How to write a Recursive Function?

Before writing a recursive function for a problem its necessary to define the solution of the problem in terms of a similar type of a smaller problem.

Two main steps in writing recursive function are as follows:

(i). Identify the Non-Recursive part(base case) of the problem and its solution(Part of the problem whose solution can be achieved without recursion).

(ii). Identify the Recursive part(general case) of the problem(Part of the problem where recursive call will be made).

Identification of Non-Recursive part of the problem is mandatory because without it the function will keep on calling itself resulting in infinite recursion

How control flows in successive recursive calls?

Flow of control in successive recursive calls can be demonstrated in following example:

Consider the following program which uses recursive function to compute the factorial of a number.

```
void main()
{
        int n,f;
        printf("Enter a number");
        scanf("%d",&amp;amp;n);
        f=fact(a);
        printf("Factorial of %d is %d",n,f);
}
int fact(int m)
{
        int a;
        if (m==1)
                return (1);
        else
                a=m*fact(m-1);
        return (a);
}
```

In the above program if the value entered by the user is 1 i.e.n=1, then the value of n is copied into m. Since the value of m is 1 the condition 'if(m==1)' is satisfied and hence 1 is returned through return statement i.e. factorial of 1 is 1. When the number entered is 2 i.e. n=2, the value of n is copied into m. Then in function fact(), the condition 'if(m==1)' fails so we encounter the statement a=m*fact(m-1); and here we meet recursion. Since the value of m is 2 the expression (m*fact(m-1)) is evaluated to (2*fact(1)) and the result is stored in a(factorial of a). Since value returned by fact(1) is 1 so the above expression reduced to (2*1) or simply 2. Thus the expression m*fact(m-1) is evaluated to 2 and stored in a and returned to main(). Where it will print 'Factorial of 2 is 2'. In the above program if n=4 then main() will

call fact(4) and fact(4) will send back the computed value i.e. f to main(). But before sending back to main() fact(4) will call fact(4-1) i.e. fact(3) and wait for a value to bereturned by fact(3). Similarly fact(3) will call fact(2) and so on. This series of calls continues until m becomes 1 and fact(1) is called, which returns a value which is so called as termination condition. So we can now say what happened for n=4 is as follows

fact(4) returns (4*fact(3))

fact(3) returns (3*fact(2))

fact(2) returns (2*fact(1))

fact(1) returns (1)

Macro substitution

In C programming, a **macro** is a symbolic name or constant that represents a value, expression, or code snippet. They are defined using the #define directive, and when encountered, the preprocessor substitutes it with its defined content.

Eg:- #define c 299792458 // speed of light

Macros in C are very useful at multiple places to replace the piece of code with a single value of the macro.

Example program

Write a c program to find area of circle

```
#include<stdio.h>
#define PI 3.142
int main()
{
        int r,area;
        printf("Enter value of radius:");
        scanf("%d", &r);
        area=PI*r*r;
        printf("Area= %d", area);
        return 0;
}
```

Command line arguments

The most important function of C/C++ is main() function. It is mostly defined with a

return type of int and without parameters.

```
int main()
{
…..
}
```

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems. To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[])
{ /* ... */ }
```

- argc (ARGument Count) is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name).
- The value of argc should be non negative.
- argv(ARGument Vector) is array of character pointers listing all the arguments.
- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments

Properties of Command Line Arguments:

1. They are passed to main() function.

2. They are parameters/arguments supplied to the program when it is invoked.

3. They are used to control program from outside instead of hard coding those values inside the code.

4. argv[argc] is a NULL pointer.

5. argv[0] holds the name of the program.

6. argv[1] points to the first command line argument and argv[n] points last argument.

C Program to add 2 numbers using command line arguments

```
#include<stdio.h>
```

```
void main(int argc, char * argv[])
{
        int i, sum = 0;
        if (argc != 3)
        {
        printf("You have forgot to type numbers");
        exit(1);
        }
        printf("The sum is :" );
        for (i = 1; i<argc; i++)
        {
                sum = sum + atoi(argv[i]);
        }
        printf("%d", sum);
}
```

STORAGE CLASS

Storage class defines the scope(visibility) and life time of variables and functions within a C program.

Storage classes help us to trace the existence of a particular variable during the runtime of a program.

There are four different storage classes in C

1. **Auto(Default storage class for local variables)**
   - This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language.
   - Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope).
   - Ofcourse, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared.
   - **int a** and **auto int a** defines the variable a in same storage class
2. **Register**

- This storage class declares register variables that have the same functionality as that of the auto variables.
- The only difference is that the compiler tries to store these variables in the register of the microprocessor rather than RAM if a free register is available.
- This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.
- Eg: register int a;

3. Static

- This storage class is used to declare static variables which are popularly used while writing programs in C language.
- Static variables have the property of preserving their value even after they are out of their scope!
- Hence, static variables preserve the value of their last use in their scope.
- So we can say that they are initialized only once and exist till the termination of the program.

Eg:
```
void test()
{
        static int t=0;
        printf("%d\n",t);
        t=t+1;
}
int main()
{
        test();
        test();
        return 0;
}
```

Output

0

1

4. Extern

- Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used.
- So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere.
- Default storage class for global variable is extern.
- The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

Storage Classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	Stack	Garbage	Within block	End of block
extern	Data segment	Zero	Global, Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

LIFE TIME AND VISIBILITY OF VARIABLE

Scope, Visibility and Lifetime of variables in C Language are very much related to each other.

Scope determines the region in a C program where a variable is available to use.

Visibility of a variable is related to the accessibility of a variable in a particular scope of the program.

Lifetime of a variable is for how much time a variable remains in the system's memory

The variables may broadly categorized , depending upon the place of their declaration as: internal(local) or External(global).

- The internal(local) variables are those which are declared inside the function.
- The external(global) variables are those which are declared outside the function.

STRUCTURE

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds. So structure is called as Heterogeneous aggregate. Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

Defining a Structure

- To define a structure, we use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows –

```
struct structure_name
{
        datatype member 1;
        datatype member 2;
        ….
        datatype member n;
};
```

Here struct is the keyword used to create structure datatype variable. Structure name is the name of the newly created structure datatype. Member 1, member 2…etc are the data members in the structure datatype.

Eg:-

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

Declare Structure variables

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

Eg :-

```
struct Point
{
        int x, y;
} p1;
```

OR

```
struct Point

{
  int x, y;
};

int main()
{
  struct Point p1;
}
```

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type.

Syntax:-student.rollno represents rollno filed of a student structure

Eg- Program using structure to store details of a student

```
#include <stdio.h>
struct student
{
   char name[50];
   int roll;
   float marks;
} s;

int main()
{
        printf("Enter information:\n");
        printf("Enter name: ");
        scanf("%s",s.name)
        printf("Enter roll number: ");
        scanf("%d", &s.roll);
        printf("Enter marks: ");
        scanf("%f", &s.marks);
```

```
    printf("Displaying Information:\n");
    printf("Name: ");
    printf("%s", s.name);
    printf("Roll number: %d\n", s.roll);
    printf("Marks: %.1f\n", s.marks);
    return 0;
}
```

Array of structure

An array of structure in C programming is a collection of different datatype variables, grouped together under a single name. To declare an array of structure, first the structure must be defined and then an array variable of that type should be defined.

For Example − struct book b[10]; //10 elements in an array of structures of type 'book'

Eg:
```
struct employee
    {
        int emp_id;
        char name[20];
        char dept[20];
        float  salary;
    };
```
Then an array of structure can be created like:

structemployeeemp[10];

In this example, the first structure **employee** is declared, then the array of structure created using a new type i.e. **struct employee**. Using the above array of structure 10 set of employee records can be stored and manipulated. To access any structure, index is used. For example, to read the emp_id of the first structure we use **scanf("%d", emp[0].emp_id);** as we know in C array indexing starts from 0.

Similar array of structure can also be declared like:

```
Struct employee
{
    int emp_id;
    char name[20];
    char dept[20];
    float  salary;
```

}emp[10];

UNION DATATYPE

Union in C is **a special data type available in C that allows storing different data types in the same memory location**. It is also a heterogeneous datatype like structure. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

```
union union_name
{
        datatype member 1;
        datatype member 2;
        ….
        datatype member n;
};
```

To declare union variable use variable name with union definition or in the main program.

In main program we can declare union **union_namevariablename;**

Eg:- union student s;

Accessing union members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access.

Eg:-s.rollno can represent the rollnumber of a student.

Example program

```
#include <stdio.h>
union student
{
  int rollNo;
  char name[32];
  double marks;
};
int main()
{
```

```
    union student s;
    printf("Size of the Union = %d bytes\n", sizeof(union student));
    printf("Enter Name:- ");
    scanf("%s",s.name);
    printf("Student name is %s\n",s.name);
    printf("Enter Roll no:- ");
    scanf("%d",&s.rollNo);
    printf("Student roll no is %d\n",s.rollNo);
    printf("Enter Marks:- ");
    scanf("%lf",&s.marks);
    printf("Student percentage is %g\n",s.marks);
    //Print Other members
    printf("Student name is %s\n",s.name);
    printf("Student roll no is %d\n",s.rollNo);
    return 0;
}
```

Program to check prime or not using function.

```
#include <stdio.h>
int prime(int);
int main()
{
int num,p;
printf("ENTER A NUMBER:");
scanf("%d",&num);
p=prime(num);
if(p==1)
printf("\n%d IS A NOT PRIME NUMBER",num);
else
printf("\n%d IS A PRIME NUMBER",num);
return 0;
}
int prime(int n)
{
int i,res=0;
```

```
for(i=2;i<=n/2;i++)
{
if(n%i==0)
{
res=1;
break;
}
else
res=0;

}
return res;
}
```