# EST 102 CPC
# MODULE-4

## WORKING WITH FUNCTIONS

# Introduction to Functions:

- C functions is one of the strengths of C language.
- **A function is a self-contained block of code that performs a particular task.**
- Once a function has been designed and packed, it can be treated as a blackbox that takes some data from the main program and returns a value.
- The inner details of operation are invisible to the rest of the program.
- All that a program knows about a function is: what goes in and what comes out.
- Functions in C can be classified into two;
  - **Library functions**: Not required to be written by the user.
    - Eg:printf, scanf, strcat etc.,
  - **User-defined functions**: developed by user at the time of writing a program.
    - Eg:main

# Need for user-defined functions:

- Even though it is possible to code any program utilizing only main function, it leads to a number of problems.
  - The program may become too large and complex.
  - Debugging, testing and maintaining becomes difficult.
- If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit.
- These independently coded programs are called *subprograms* that are much easier to understand, debug and test.
- Such subprograms are called ***functions.***

# 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**.
- These modules are carefully integrated to become a software system that satisfies the system requirements.
- **Characteristics of Modular Programming:**
  - Each module should do only one thing.
  - Communication between modules is allowed only by calling a module.
  - A module can be called only by one and only one higher module.
  - No communication can take place directly between modules that do not have calling-called relationship.
  - All  modules are designed as single-entry, single-exit systems using control structures.

# Writing Functions:

- **Eg:C program to print a message using function.**

```
#include<stdio.h>
void main( )
{
printf(“This is the main function”);
printmessage( );
}
void printmessage( )
{
printf(“This is user defined function”);
}
```

**OUTPUT:**

This is main Function!
This is user defined function

- The program execution begins with the main function.
- During execution of the main, after execution of printf statement, the following statement is encountered,

```
printmessage( );
```

  which indicates that the function printmessage is to be executed.

- At this point, the program control is transferred to the function printmessage.
- printmessage function is executed and then the main function execution ends.

# Elements of user defined functions:

- In order to make use of a user-defined function, we need to establish three elements that are related to functions.
  - Function definition.
  - Function call.
  - Function declaration.
- The **function definition** is an independent program module that is specially written to implement the requirements of the function.
- In order to use this function, we need to invoke it a required place in the program. This is **function call.**
- The program (or a function) that calls the function is referred to as the calling program or calling function.
- The calling program should declare any function that is to be used later in the program. This is called **function declaration** or function prototype.

## Function Definition:

- A function definition include the following;

1. Function name.
2. Function type.
3. List of parameters.
4. Local variable declarations.
5. Function statements
6. Return statement

- All these 6 elements are grouped into two parts;
  - Function header (First three elements)
  - Function body (Second three elements)

```
function_type  function_name(parameter list)
   {
      local variable declaration;
      executable statement1;
      executable statement2;
      . . . . .
      . . . . .
      return statement;
   }
```

- The first line function_type function_name(parameter list) is known as the function header and the statements within the opening and closing braces constitute the function body, which is a compound statement.

## Function Header:

- It has three parts;
  - The function type
  - The function name
  - The formal parameter list

## Name and Type:

- **The function type** specifies the type of value that the function is expected to return to the program calling the function.
- If the return type is not explicitly specified, C will assume that it is an integer type.
- If the function is not returning anything, Then we need to specify the return type as **void.**
- The value returned is the output produced by the function.
- It is a good programming practice to explicitly specify the return type even when it is an integer.
- The function name is any valid C identifier and therefore must follow the same rules of formation as other variable names in C.
- The name must be appropriate to the task performed by the function.

## Formal Parameter List:

- It declares the variables that receive the data sent by the calling program.
- They serve as the input data to the function to carry out the specified task.
- Since they represent the actual input values, they are called formal parameters.
- These parameters can be used to send values to the calling programs.
- The parameters are also called arguments.
- The parameter list contains variable declarations separated by commas and are surrounded by parentheses.
- Eg: float quadratic (int a, int b, int c) {..... }

  int add(int a, int b) {....}

- A function need not receive values from the calling program.
- In such cases, function do not have formal parameters.
- To indicate empty parameter list,  we use the term void as below,

```
void printmessage (void)
{
.....
}
```

## Function Body:

- Contains the declarations and statements necessary for performing the required task.
- The body is enclosed in braces and has 3 parts;
  - **Local variable declarations**: Specify the variables needed by the function.
    - They have no role in communication between the functions.
  - **Function statements**: Performs the task of the function.
  - **A return statement**: Returns the value evaluated by the function.
    - It can be omitted if the function does not return any value (In such cases, its return type should be specified as void).
    - When a function reaches its return statement, the control is transferred back to the calling program.
    - In the absence of a return statement, the closing brace act as a void return.

## Return values and their types:

- The return statement can take one of the following forms;

  **return;**

  or

  **return(expression);**

- The first return does not return any value. It acts much as the closing brace of the function.
  - Eg: **if(error)**

    **return;**

- The control is immediately passed back to the calling function when a return is encountered.

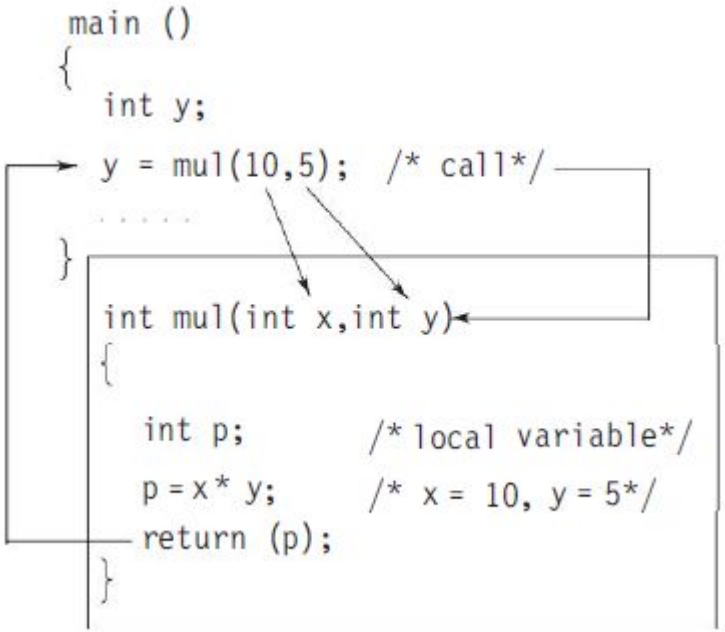
- The second form of return with an expression returns the value of the expression.
    - Eg: int mul (int x, int y) //This fn. returns the value of p which is the product of the values of x and y.

```
{
   int p;
   p=x*y;
   return(p);
}
```

- **All functions by default return int type data.**
- By using a *type specifier* in the function header, we can force a function to return a particular type of data. When a value is returned, it is automatically cast to the function's type.

# Function Calls:

- **A function can be called by simply using the function name followed by a list of actual parameters( or arguments), if any, enclosed in parentheses.**
- Eg:




- When the compiler encounters the function call, the control is transferred to the function mul( ).
- This function is then executed line-by-line and a value is returned upon reaching the return statement.
- The function call sends two integer values 10 and 5 to the function which are assigned to x and y respectively.
- The function computes the product x and y, assigns the result to the local variable p, and then return the value 50 to the main where it is assigned to y again.

## **Function Declaration:**( also known as ***Function Prototype***)

- All functions in a C program must be declared, before they are invoked.
- It has 4 parts;
  - Function type( return type)
  - Function name.
  - Parameter list.
  - Terminating semicolon.
- They are coded in the following format;

  ***function_type function_name (parameter list);***

- Eg:

  int mul(int m, int n);

- In a C program, a prototype definition can be placed in two places;

1. **Above all the functions** (including *main*)

- When the declaration is placed above all the functions (in the global declaration section), the prototype is called *global prototype*.
- Such declarations are available for all the functions in the program.

2. **Inside a function definition.**

- When the declaration is placed in a function definition (in the local declaration section), the prototype is called *local prototype*.
- Such declarations are primarily used by the functions containing them.
- The place of declaration of a function defines a region in a program in which a function may be used by other functions.
- This region is called **scope** of the function.
- It is a good programming style to declare prototypes in the global declaration section before main. It adds flexibility, provides an excellent quick reference to the functions used in the program and enhances documentation.

## Actual v/s Formal parameters:

- Parameters are used in all the three elements of a user defined function such as;
  - Function declaration.
  - Function definition.
  - Function call.
- The parameters used in function declarations and function definitions are called formal parameters.
- The parameters that are used in function calls are called actual parameters.They can be simple constants, variables or expressions.
- The formal and actual parameters must match exactly in type, number and order. Their names do not need to match.

# Pass by Value:

- **Parameter passing:** The technique used to pass data from one function to another.
- This can be done in two ways:
  - **Pass by value** (also known as call by value)
  - **Pass by pointers** (also known as call by pointers)

**Pass by value** :

- In pass by value, values of actual parameters are copied to the variables in the parameters list of the called function.
- The called function works on the copy and not on the original values of the actual parameters.
- This ensures that the original data in the calling function cannot be changed accidentally. We can not modify the value of the actual parameter by using the formal parameter.

- Consider the following example;

```c
/*Find SUM of 2 int numbers using User Defined Functions*/
#include <stdio.h>
int sumTwoNum(int, int); /*function declaration*/
int main()
{
    int number1, number2;
    int sum;

    printf("Enter the first integer number: ");
    scanf("%d", &number1);

    printf("Enter the second integer number: ");
    scanf("%d", &number2);

    sum = sumTwoNum(number1, number2); //function call
    printf("Number1: %d, Number2: %d\n", number1, number2);
    printf("Sum: %d\n", sum);
    return 0;
}
int sumTwoNum(int x, int y) /*function definition*/
{
    /*x and y are the formal parameters*/
    int sum;
    sum = x + y;
    return sum;
}
```

- Here, the actual parameters are number1 and number2.
- These values are copied to the parameters x and y respectively.
- So, it is impossible to modify the value of the actual parameter by using the formal parameter.
- The changes made to the formal parameters in the called function have no effect on the values of actual variables in the calling function.

## Pass by pointers:

- In pass by pointers, **the memory address of the variables are sent to the called function** rather than the copies of the values.
- So, the address of the actual variables in the calling function are copied to the variables of the called function.
- Here, the called function directly works on the data in the calling function and the changed values are available in the calling function for its use.
- Used commonly for array and string manipulations.
- This method is also used when we require multiple values to be returned by the called function.

# Recursion:

- In C language, **a function is allowed to call itself**. This is called recursion.
- When a called function in turn calls another function, a process of ‘chaining’ occurs. Recursion is a special case of this process.
- While using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.
- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.
- Eg:

```
main( )
    {
        printf(“This is an example of recursion\n”);
        main( );
    }
```

- Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem.
- When we write recursive functions, we must have an if statement somewhere to force the function to return without the recursive call being executed.
- Otherwise the function will never return.

# Factorial using Recursion:

```c
/*Factorial using recursion*/
#include <stdio.h>
long factorial(int);

int main()
{
   int n;
   long f;
   printf("Enter a positive integer:\n");
   scanf("%d",&n);
   f=factorial(n);
   printf("%d!=%ld",n,f);
   return 0;
}

long factorial(int n)
{
    if(n==0)
    return 1;
    else
    return(n*factorial(n-1));
}
```

# The scope & lifetime of variables:

- In addition to a data type, variables in C do have a **storage class**.
- Following are the important storage classes in C;
  - **Automatic variables**
  - **External variables**
  - **Static variables**
  - **Register variables**
- Storage classes provide information about variable's location and visibility.
- Variable's storage class tells us
  - Where the variable would be stored
  - What will be the initial value of the variable(ie,the default initial value)
  - What is the scope of the variable(ie,in which functions the value of the variable would be available)
  - What is the lifetime of a variable(ie,how long would the variable exist)

- Variables can be broadly classified into two depending on the place of their declaration;
  - **Local variables** (Internal variables):declared within a particular function.
  - **Global variables** (External variables): declared outside the function.

## Automatic Variables:

- **Declared inside a function in which they are to be utilized**.
- Created when the function is called and destroyed automatically when the function is exited.
- Such variables are private(or local) to the function in which they are declared.
  - So, they are also called **local or internal variables**.
- A variable declared inside a function without storage class specification is, by default, an automatic variable.

- To explicitly declare automatic variable, we can use the keyword **auto**.
  - Eg:

```
main( )
{
    auto int number;
    ……..
}
```

- Value of an automatic variable cannot be changed accidentally by what happens in some other function in the program. So, we may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

## External Variables:

- Variables that are **alive and active throughout the entire program**.
- Also called **global variables**.
- External variables can be accessed by any function in the program.
- They are declared outside the function.
- Once a variable has been declared as global, any function can use it and change its value. Then subsequent functions can reference only that new value.
- To explicitly declare a variable as an external variable, we can use the keyword **extern.**
    - Eg1: extern int y;

- Eg2:The external declaration of integer ***number*** and float ***length*** may appear as:

```
int number;
float length = 7.5;
main( )
{
  --------
  --------
}
function1( )
{
  --------
  --------
}
function2( )
{
  --------
  --------
}
```

- Here, the variables ***number*** and ***length*** are available to use for all the three functions main( ),function1( ) and function2( ).

- Eg3:In certain cases, the **global and local variable may have the same name**. In such cases, **local variable will have precedence over the global one** in the function where it is declared.

```
int count;
main( )
{
    count = 10;
    -----
    -----
}
function( )
{
int count = 0;
     -----
     -----
   count = count+1;
}
```

- When the ***function( )*** references the variable ***count***, it will be referencing only its local variable, not the global one.
- The value of ***count*** in ***main( )*** will not be affected.

## Static Variables:

- The **value of static variable 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 external type depending on the place of declaration.
- It is also possible to control the scope of a function.
    - Eg: If We would like a particular function accessible only to the functions in the file in which it is defined, and not to any function in other files.
    - This can be accomplished by defining that function with the storage class **static**.

## Internal static variables:

- Declared inside a function.
- Scope extends up to the end of the function in which they are defined.ie,they are similar to *auto* variables except the fact that they remain in existence throughout the remainder of the program.
- Therefore they are used to retain values between function calls.

## External static variables:

- Declared outside all functions and is available to all functions in that program.
- The difference between a static external variable and a simple external variable is that static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

## Register Variables:

- To 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 much faster than a memory access, keeping 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 kept in the register, it is important to carefully select the variables for this purpose.
- However, once the limit is reached, C will automatically convert register variables into non-register variables.

| *Storage Class* | *Where declared* | *Visibility (Active)* | *Lifetime (Alive)* |
|---|---|---|---|
| None | Before all functions in a file (may be initialized) | Entire file plus other files where variable is declared with **extern** | Entire program (Global) |
| **extern** | Before all functions in a file (cannot be initialized)<br>**extern** and the file where originally declared as global. | Entire file plus other files where variable is declared | Global |
| **static** | Before all functions in a file | Only in that file | Global |
| None or **auto** | Inside a function (or a block) | Only in that function or block | Until end of function or block |
| **register** | Inside a function or block | Only in that function or block | Until end of function or block |
| **static** | Inside a function | Only in that function | Global |

# Structures:

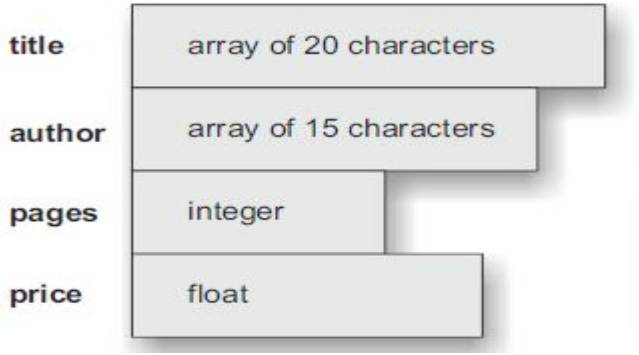
- **Collection of variables of different data types that are referenced by a common name.**
- Structure is a **constructed data type** in C.
- It helps to pack data of different types.
- It acts as a convenient tool to handle a group of logically related data items.
- Structures help to organize complex data in a more meaningful way.
- A structure contains a number of data types grouped together.
- These data types may or may not be of the same type.

## Defining a structure:

- Structures must be defined first before using them.
- Eg:Consider a book database consisting of ;
  - The book name
  - Author
  - No of pages
  - Price
- To hold this information,we can define a structure as follows;

```
struct book_bank
{
      char      title[20];
      char      author[15];
      int       pages;
      float     price;
};
```

- The keyword **struct** declares a structure to hold the details of four data fields, namely title, author, pages and price.
- These fields are called **structure elements** or **members.**
- Each member may belong to a different type of data.
- book_bank is the name of the structure and is called the **structure tag.**
- The tag name may be used subsequently to declare the variable's that have the tag's structure.
- The structure definition has not declared any variables. It simply describes a format called template to represent information as shown below:



- The general format of a structure definition is as follows:

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

## Declaring Structure Variables:

- 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 types.

- It includes the following elements;
  - The keyword struct
  - The structure tag_name
  - List of variable names separated by commas
  - A terminating semicolon
- Eg:

```
struct book_bank
{
        char    title[20];
        char    author[15];
        int     pages;
        float   price;
};
struct book_bank book1, book2, book3;
```

- The use of tag_name is optional here.

- The statement
  struct book_bankbook1,book2,book3;
  declares book1, book2 and book3 as variables of type struct book_bank.
- Each one of these variables has 4 members as specified by the template.
- The members of structures themselves are not variables.
- They won't occupy any memory until they are associated with structure variables such as book1.

## Accessing structure members:

- To access members of a structure, we can use a **member operator ' . '** (or structure operator or period operator) to establish the link between a variable and a member.
- Eg:

book1.price

is the variable representing the price of book1 and can be treated like any other ordinary variable.

- To assign values to the members of book1;

```
book1.pages = 500;

book1.price = 250;

strcpy(book1.title, "C BASICS");
```

- To give values via keyboard, we can use

```
scanf(“%s\n”,book1.title);

scanf(“%d\n”,&book1.pages);
```

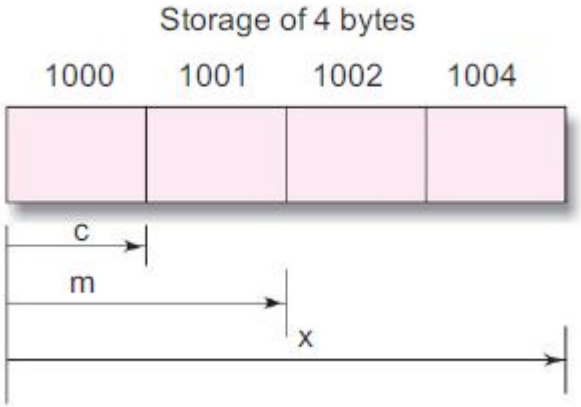
# Union:

- Unions are concept borrowed from structures.
- Therefore, union follow the same syntax as structures.
- The major distinction between structure and union is in terms of storage.
- In structure, each member has its own storage location, whereas all the members of a union use the same location.
- This implies that, although a union may contain many members of different types, it can handle only one member at a time.
- A union is declared by using the keyword **union**.
- Eg:

```
union item
{
    int m;
    float x;
    char c;
} code;
```

- This declares a variable *code* of type union item.
- The union has 3 members, each with different data type.
- Only one of them can be used at a time.

- Only one location is allocated for a union variable, irrespective of its size.
- Sharing of storage location by union members;




- The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.
- In the example taken, the member x requires 4 bytes which is the largest among the members.
- Thus, as shown in figure, all the three variables shares the same address.

- To access the union variable, we can use the same syntax as that we used for structure variables.

  Eg: code.m

  code.x

- Unions can be used in all places where structure is allowed.
- While initializing unions, care must be taken to initialize it only with value of the same type as the first union member.
- In our example,

  union item abc = {100}; is valid

  union item abc = { 10.2} is invalid because the first member is of type int.