LEARNING OBJECTIVE

Till now, we have heard the word 'function' several times and we know that main(), printf(), and scanf() are the functions that we have been using in every program. Basically, C supports two types of functions: user-defined functions and library functions. While main() is a user-defined function, printf() and scanf() are library functions.

The difference between a library function and a user-defined function is that we as programmers do not have to write the code to implement a library function whereas we need to code the user-defined function to perform a well-defined task. In this chapter, we will explore functions and their importance in detail.

10.1 INTRODUCTION

C enables programmers to break up a program into segments commonly known as *functions*, each of which can be written more or less independently of the others. Every function in the program is supposed to perform a well-defined task. Therefore, the program code of one function is completely insulated from the other functions.

Every function interfaces to the outside world in terms of how information is transferred to it and how results generated by the function are transmitted back from it. This interface is basically specified by the function name. For example, look at Figure 10.1 which explains how the `main()` function calls another function to perform a well-defined task.

From the figure we see that `main()` calls the function named `func1()`. Therefore, `main()` is known as the *calling function* and `func1()` is known as the *called function*. The moment the compiler encounters a function call, instead

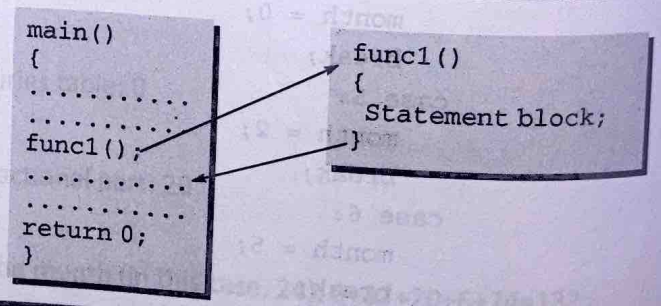


Figure 10.1 The function main() calls func1()

of executing the next statement in the calling function, the control jumps to the statements that are a part of the called function. After the called function is executed, the control is returned back to the calling program.

It is not necessary that the `main()` function can call only one function; it can call as many functions as it wants and as many times as it wants. For example, a function call placed within a `for` loop, `while` loop, or `do-while`

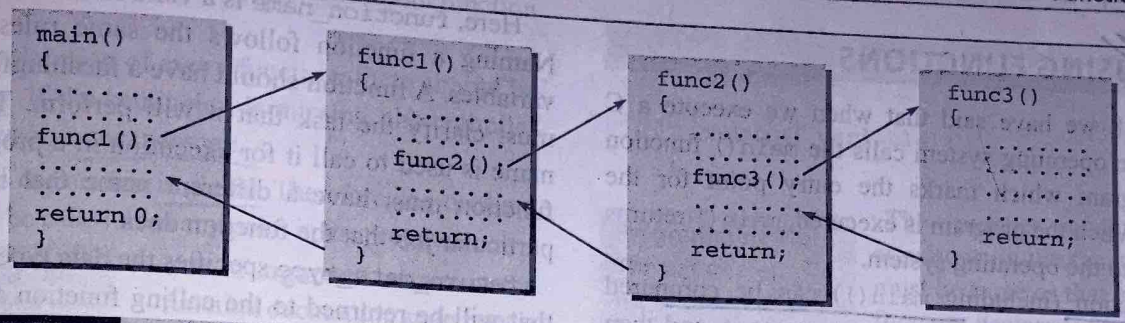


Figure 10.2 Function calling another function

loop may call the same function multiple times until the condition holds true.

Another point is that it is not only the `main()` function that can call other functions. A function can call any other function. For example, look at Figure 10.2 which shows one function calling another, and this function in turn calling some other function. From this we see that every function encapsulates a set of operations and when called it returns information to the calling program.

10.1.1 Need for Functions

Let us analyse the reasons for segmenting a program into manageable chunks as it is an important aspect to programming.

- Dividing the program into separate well-defined functions facilitates each function to be written and tested separately. This simplifies the process of getting the total program to work. Figure 10.3 shows that `main()` calls other functions for dividing the entire code into smaller sections (or functions). This approach is referred to as the *top-down* approach.

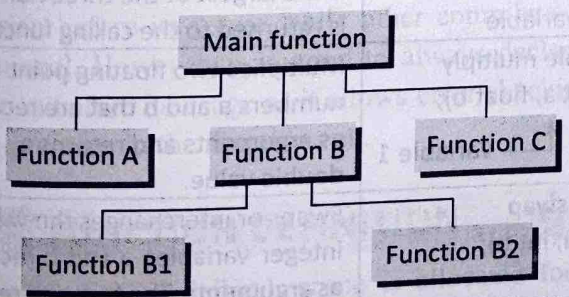


Figure 10.3 Top-down approach of solving a problem

- Understanding, coding, and testing multiple separate functions are far easier than doing the same for one huge function.
- If a big program has to be developed without the use of any function other than `main()`, then there will be countless lines in the `main()` function and maintaining this program will be very difficult. A large program size is a serious issue in micro-computers where memory space is limited.
- All the libraries in C contain a set of functions that the programmers are free to use in their programs. These functions have been pre-written and pre-tested, so the programmers can use them without worrying about their code details. This speeds up program development, by allowing the programmer to concentrate only on the code that he/she has to write.
- When a big program is broken into comparatively smaller functions, then different programmers working on that project can divide the workload by writing different functions.
- Similar to C libraries, programmers can also write their functions and use them from different points in the main program or any other program that needs its functionalities.

Consider a program that executes a set of instructions repeatedly *n* times, though not continuously. In case the instructions had to be repeated continuously for *n* times, they can be placed within a loop. However, if these instructions have to be executed abruptly from anywhere within the program code, then instead of writing them wherever they are required, a better idea is to place these instructions in a function and call that function wherever required. Figure 10.4 explains this concept.

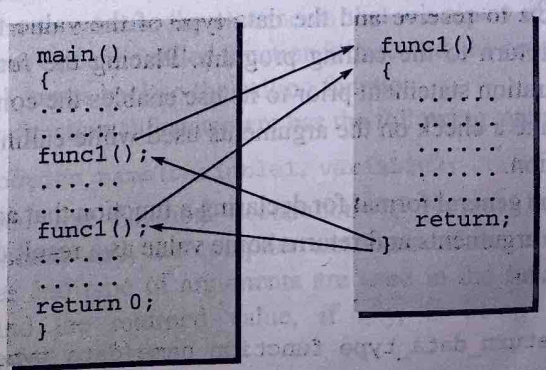


Figure 10.4 Function func1() called twice from main()

10.2 USING FUNCTIONS

In Chapter 8 we have said that when we execute a C program, the operating system calls the `main()` function of the program which marks the entry point for the execution. When the program is executed, `main()` returns some value to the operating system.

Any function (including `main()`) can be compared to a black box that takes in input, processes it, and then produces the result. However, we may also have a function that does not take any inputs at all, or one that does not return anything at all.

While using functions we will be using the following terminologies:

- A function `f()` that uses another function `g()`, is known as the *calling function* and `g()` is known as the *called function.*
- The inputs that the function takes are known as *arguments/parameters.*
- When a called function returns some result back to the calling function, it is said to *return* that result.
- The calling function may or may not pass *parameters* to the called function. If the called function accepts arguments, the calling function will pass parameters, else it will not do so.
- *Function declaration* is a declaration statement that identifies a function with its name, a list of arguments that it accepts, and the type of data it returns.
- *Function definition* consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

10.3 FUNCTION DECLARATION/ FUNCTION PROTOTYPE

Before using the function, the compiler must know about the number and type of parameters that the function expects to receive and the data type of the value that it will return to the calling program. Placing the function declaration statement prior to its use enables the compiler to make a check on the arguments used while calling that function.

The general format for declaring a function that accepts some arguments and returns some value as a result can be given as:

```
return_data_type function_name(data_type
   variable1, data_type variable2,...);
```

Here, `function_name` is a valid name for the function. Naming a function follows the same rules as naming variables. A function should have a meaningful name that must clarify the task that it will perform. The function name is used to call it for execution in a program. Every function must have a different name that indicates the particular job that the function does.

`return_data_type` specifies the data type of the value that will be returned to the calling function as a result of the processing performed by the called function.

`data_type variable1, data_type variable2, ...` is a list of variables of specified data types. These variables are passed from the calling function to the called function. They are also known as *arguments* or *parameters* that the called function accepts to perform its task. Table 10.1 shows examples of valid function declarations in C.

Table 10.1 Valid function declarations

Function declaration	Use of the function
Return data type ↓ char convert_to_ uppercase (char ch);	Converts a character to upper case. The function receives a character as an argument, converts it into upper case, and returns the converted character back to the calling program.
Function name ↓ float avg (int a, int b);	Calculates the average of two numbers a and b received as arguments. The function returns a floating point value.
int find_largest (int a, int b, int c); ↑ Data type of variable	Finds the largest of three numbers— a, b, and c received as arguments. An integer value which is the largest of the three numbers is returned to the calling function.
double multiply (float a, float b); ↑ Variable 1	Multiplies two floating point numbers a and b that are received as arguments and returns a double value.
void swap (int a, int b);	Swaps or interchanges the value of integer variables a and b received as arguments. The function returns no value, therefore the data type is void.
void print(void);	The function is used to print information on screen. The function neither accepts any value as argument nor returns any value. Therefore, the return type is void and the argument list contains void data type.

The following points should be kept in mind about function declaration:

- After the declaration of every function, there should be a semicolon. If the semicolon is missing, the compiler will generate an error message.
- The function declaration is global. Therefore, the declared function can be called from any point in the program.

Programming Tip: Though optional, use argument names in the function declaration.

- Use of argument names in the function declaration statement is optional. So both the following declaration statements are valid in C:

```
int func(int, char, float);
int func(int num, char ch, float fnum);
```

- A function cannot be declared within the body of another function.
- A function having void as its return type cannot return any value.
- A function having void as its parameter list cannot accept any value. So the function declared as

```
void print(void); or void print();
```

does not accept any input/arguments from the calling function.

- If the function declaration does not specify any return type, then by default, the function returns an integer value. Therefore when a function is declared as

```
sum(int a, int b);
```

Then the function `sum()` accepts two integer values from the calling function and in turn returns an integer value to the caller.

- Some compilers make it compulsory to declare the function before its usage while other compilers make it optional. However, it is good to always declare the function before its usage as it allows error checking on the arguments in the function call.

10.4 FUNCTION DEFINITION

When a function is defined, space is allocated for that function in the memory. A function definition comprises two parts:

Programming Tip: It is an error to place a semicolon after the function header in the function definition.

- Function header
- Function body

The syntax of a function definition can be given as:

```
return_data_type function_name(data_type variable1, data_type variable2,...)
{
   ...
   statements
   ...
   return( variable);
}
```

Programming Tip: The parameter list in the function definition as well as function declaration must match.

The number and order of arguments in the function header must be same as that given in the function declaration statement.

While `return_data_type function_name(data_type variable1, data_type variable2,...)` is known as the function header, the rest of the portion comprising program statements within { } is the function body which contains the code to perform the specific task.

The function header is same as that of the function declaration. The only difference between the two is that a function header is not followed by a semicolon. The list of variables in the function header is known as the *formal parameter list*. The parameter list may have zero or more parameters of any data type. The function body contains instructions to perform the desired computation in a function.

Programming Tip: A function can be defined either before or after the main().

The function definition itself can act as an implicit function declaration. So the programmer may skip the function declaration statement in case the function is defined before being used.

Note The argument names in the function declaration and function definition need not be the same. However, the data types of the arguments must match with that specified in function declaration as well as function definition.

10.5 FUNCTION CALL

The function call statement invokes the function. When a function is invoked the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function.

The function call statement has the following syntax:

```
function_name(variable1, variable2, ...);
```

When the function declaration is present before the function call, the compiler can check if the correct number and type of arguments are used in the function call and the returned value, if any, is being used reasonably.

Function definitions are often placed in a separate header file which can be included in other C source files

that wish to use the functions. For example, the header file `stdio.h`, contains the definition of `scanf` and `printf` functions. We simply include this header file and call these functions without worrying about the code to implement their functionality.

Note: The list of variables used in a function call is known as the actual parameter list. The actual parameter list may contain variable names, expressions, or constants.

10.5.1 Points to Remember While Calling a Function

The following points are to be kept in mind while calling a function:

- Function name and the number and type of arguments in the function call must be same as that given in the function declaration and the header of the function definition.

Programming Tip: A logic error will be generated if the arguments in the function call are placed in the wrong order.

- If by mistake, the parameters passed to a function are more than what it is specified to accept, then the extra arguments will be discarded.
- If by mistake, the parameters passed to a function are less than what it is specified to accept, then the unmatched argument will be initialized to some garbage value.
- Names (and not the types) of variables in function declaration, function call, and header of function definition may vary.
- If the data type of the argument passed does not match with that specified in the function declaration, then either the unmatched argument will be initialized to some garbage value or compile time error will be generated.
- Arguments may be passed in the form of expressions to the called function. In such cases, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.
- The parameter list must be separated with commas.
- If the return type of the function is not void, then the value returned by the called function may be assigned to some variable as shown in the following statement.

```
variable_name = function_name(variable1,
   variable2, ...);
```

Let us now try writing a program using functions.

1. Write a program to add two integers using functions.

```
#include <stdio.h>
// FUNCTION DECLARATION
int sum(int a, int b);
int main()
{
   int num1, num2, total = 0;
   printf("\n Enter the first number: ");
   scanf("%d", &num1);
   printf("\n Enter the second number: ");
   scanf("%d", &num2);
   total = sum(num1, num2);
   // FUNCTION CALL
   printf("\n Total = %d", total);
   return 0;
}
// FUNCTION DEFINITION
int sum (int a, int b)     // FUNCTION HEADER
{  // FUNCTION BODY
   int result;
   result = a + b;
   return result;
}
```

Output

```
Enter the first number: 20
Enter the second number: 30
Total = 50
```

Programming Tip: It is an error to use void as the return type when the function is expected to return a value to the calling function.

The variables declared within the function and its parameters are local to that function. The programmer may use same names for variables in other functions. This eliminates the need for having unique names for variables declared in all the functions in the program.

In the function `sum()`, we have declared a variable result just like any other variable. Variables declared within a function are called *automatic local variables* because of two reasons.

- First, they are local to the function. So, their effect (in terms of scope, lifetime, or accessibility) is limited to the function. Any change made to these variables is visible only in that function.
- Second, they are automatically created whenever the function is called and they cease to exist at the end of the function.

Note: A function cannot be used on the left side of an assignment statement. Therefore writing, `func(10) = 100;` is invalid in C, where `func()` is a function that accepts an integer value.

10.6 RETURN STATEMENT

The return statement is used to terminate the execution of a function and returns control to the calling function. When this statement is encountered, the program execution resumes in the calling function at the point immediately following the function call. Refer Figure 10.2 in which the control passes from the called function to the calling function when the return statement is encountered.

Programming Tip: It is an error to use a return statement in a function that has void as its return type.

A return statement may or may not return a value to the calling function. The syntax of return statement can be given as

```
return <expression>;
```

Here expression is placed in between angular brackets because specifying an expression is optional. The value of expression, if present, is returned to the calling function. However, in case expression is omitted, the return value of the function is undefined.

The expression, if present, is converted to the type returned by the function. A function that has void as its return type cannot return any value to the calling function. So a function that has been declared with return type void, a return statement containing an expression generates a warning and the expression is not evaluated.

For functions that have no return statement, the control automatically returns to the calling function after the last statement of the called function is executed. In other words an implicit return takes place upon completion of the last statement of the called function, and control automatically returns to the calling function.

Note The programmer may or may not place the expression in a return statement within parentheses angular brackets. By default, the return type of a function is int.

Programming Tip: When the value returned by a function is assigned to a variable, then the returned value is converted to the type of the variable receiving it.

A function may have more than one return statement. For example, consider the following program:

```
#include <stdio.h>
#include <conio.h>
int check_relation(int a,
int b);
// FUNCTION DECLARATION
main()
{
  int a=3, b=5, res;
  clrscr();
  res = check_relation(a, b);
// FUNCTION CALL
  if(res==0) // Test the returned value
    printf("\n EQUAL");
  if(res==1)
    printf("\n a is greater than b");
  if(res==-1)
    printf("\n a is less than b");
  getch();
  return 0;
}
int check_relation(int a, int b)
// FUNCTION DEFINITION
{
  if(a==b)
    return 0;
  if(a>b)
    return 1;
  else if (a<b)
    return -1;
}
```

Programming Tip: An error is generated when the function does not return data of the specified type.

Output

```
a is less than b
```

In this program, there are multiple return statements, but only one of them will get executed depending upon the condition. The return statement, like the break statement, is used to cause a premature termination of the function.

Programming Tip: A function that does not return any value cannot be placed on the right side of the assignment operator.

An expression appearing in a return statement is converted to the return type of the function in which the statement appears. If no implicit conversion is possible, the statement is invalid. Since the return type of the function check_relation() is int, so the result of 0, 1, or -1 is evaluated as an integer.

We have mentioned earlier that the variables declared inside the function cease to exist after the last statement of the function is executed. So how can we return the value of sum in the program that adds two integers using a function? The answer to this question is that a copy of the value being returned is made automatically and this copy is available to the return point in the program.

10.6.1 Using Variable Number of Arguments

Some functions have a variable number of arguments and data types that cannot be known at the compile time. Typical examples of such functions include the printf() and scanf() functions. ANSI C offers a symbol called *ellipsis* to handle such functions. The ellipsis consists of three periods (...). It can be used as

```
int func(char ch, ...);
```

This function declaration statement is states that func() is a function that has an arbitrary number and type of argument. However, one must ensure that both the

function declaration and function definition should use the ellipsis symbol.

10.7 PASSING PARAMETERS TO THE FUNCTION

When a function is called, the calling function may have to pass some values to the called function. We have been doing this in the programming examples given so far. We will now learn the technicalities involved in passing arguments/parameters to the called function.

Basically, there are two ways in which arguments or parameters can be passed to the called function. They include:

- *Call by value* in which values of the variables are passed by the calling function to the called function. The programs that we have written so far call the function using call by value method of passing parameters.
- *Call by reference* in which address of the variables are passed by the calling function to the called function.

10.7.1 Call by Value

Till now, we had been calling functions and passing arguments to them using call by value method. In this method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task.

If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function no change will be made to the value of the variables. This is because all the changes were made to the copy of the variables and not on the actual variables.

Programming Tip: It is legal to have multiple return statements in C.

To understand this concept, consider the following code. The function `add()` accepts an integer variable `num` and adds `10` to it. In the calling function, the value of `num = 2`. In `add()`, the value of `num` is modified to `20` but in the calling function the change is not reflected.

```
#include <stdio.h>
void add(int n);    // FUNCTION DECLARATION
int main()
{
   int num = 2;
   printf("\n The value of num before
     calling the function = %d", num);
   add(num);          // FUNCTION CALL
   printf("\n The value of num after calling
     the function = %d", num);
   return 0;
}
void add(int n)  // FUNCTION DEFINITION
{
   n = n + 10;
   printf("\n The value of num in the called
     function = %d", n);
}
```

Output

```
The value of num before calling the function = 2
The value of num in the called function = 20
The value of num after calling the function = 2
```

Since the called function uses a copy of `num`, the value of `num` in the calling function remains untouched. This concept can be more clearly understood from Figure 10.5.

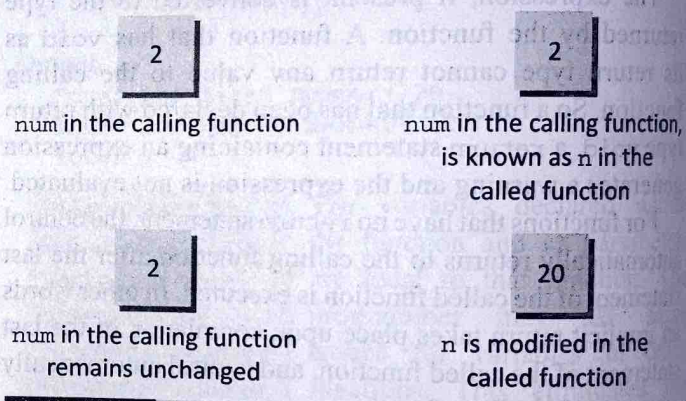


Figure 10.5 Call-by-value method of argument passing

In this program, the called function could not directly modify the value of the argument that was passed to it. In case the value has to be changed, then the programmer may use the return statement. This is shown in the following code.

```
#include <stdio.h>
int add (int n);    // FUNCTION DECLARATION
int main()
{
   int num = 2;
   printf("\n The value of num before
     calling the function = %d", num);
   num = add(num); // FUNCTION CALL
   printf("\n The value of num after calling
     the function = %d", num);
   return 0;
}

int add(int n)   // FUNCTION DEFINITION
```

```
{
   n = n + 10;
   printf("\n The value of num in the called
     function = %d", n);
   return n;
}
```

Output

```
The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 12
```

The following points are to be noted while passing arguments to a function using the call by value method.

- When arguments are passed by value, the called function creates new variables of the same data type as the arguments passed to it.
- The values of the arguments passed by the function are copied into the newly created variables.
- Arguments are called by value when the function does not need to modify the values of the original variables in the calling program.
- Values of the variables in the calling functions remain unaffected when the arguments are passed using call by value technique.

Therefore, the call by value method of passing arguments to a function must be used only in two cases:

- When the called function does not modify the value of the actual parameter. It simply uses the value of the parameters to perform its task.
- When you want that the called function should only temporarily modify the value of the variables and not permanently. So although the called function may modify the value of the variables, but these variables remain unchanged in the calling program.

Pros and Cons

Programming Tip: Using the call by value method of passing values must be preferred to avoid inadvertent changes to variables of the calling function in the called function.

The biggest advantage of using the call by value technique to pass arguments to the called function is that arguments can be variables (e.g., x), literals (e.g., 6), or expressions (e.g., $x + 1$). The disadvantage is that copying data consumes additional storage space. In addition, it can take a lot of time to copy, thereby resulting in performance penalty, especially if the function is called many times.

10.7.2 Call by Reference

When the calling function passes arguments to the called function using the call by value method, the only way to return the modified value of the argument to the caller is by explicitly using the return statement. The better option when a function wants to modify the value of the argument is to pass arguments using the call by reference technique. In call by reference, we declare the function parameters as references rather than normal variables. When this is done any changes made by the function to the arguments it receives are visible to the calling program.

To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list. This way, changes made to the parameter in the called function body will then be reflected in its value in the calling program.

Hence, in the call by reference method, a function receives an implicit reference to the argument, rather than a copy of its value. Therefore, the function can modify the value of the variable and that change will be reflected in the calling function as well. The following program uses this concept.

```
#include <stdio.h>
void add (int *n);
int main()
{
   int num = 2;
   printf("\n The value of num before
     calling the function = %d", num);
   add(&num);
   printf("\n The value of num after calling
     the function = %d", num);
   return 0;
}
void add( int *n)
{
   *n = *n + 10;
   printf("\n The value of num in the called
     function = %d", n);
}
```

Output

```
The value of num before calling the function = 2
The value of num in the called function = 20
The value of num after calling the function = 20
```

Advantages

The advantages of using the call by reference technique of passing arguments are as follows:

- Since arguments are not copied into new variables, it provides greater time- and space-efficiency.
- The function can change the value of the argument and the change is reflected in the caller.
- A function can return only one value. In case we need to return multiple values, pass those arguments by reference so that modified values are visible in the calling function.

Disadvantage

However, the side-effect of using this technique is that when an argument is passed using call by address, it

becomes difficult to tell whether that argument is meant for input, output, or both.

Now let us write a few programs that use the call by value and the call by reference mechanisms.

2. Write a function to swap the value of two variables.

```
#include <stdio.h>
void swap_call_by_val(int, int);
void swap_call_by_ref(int *, int *);
int main()
  {
     int a = 1, b = 2, c = 3, d = 4;
     printf("\n In main(), a = %d and b = %d",
        a, b);
     swap_call_by_val(a, b);
     printf("\n In main(), a = %d and b = %d",
        a, b);
     printf("\n\n In main(), c = %d and d =
        %d", c, d);
     swap_call_by_ref(&c, &d);
        // address of the variables is passed
     printf("\n In main(), c = %d and d =
        %d", c, d);
     return 0;
  }
  void swap_call_by_val(int a, int b)
  {
     int temp;
     temp = a;
     a = b;
     b = temp;
     printf("\n In function (Call By Value
        Method) a = %d and b = %d", a, b);
  }
  void swap_call_by_ref(int *c, int *d)
  {
     int temp;
     temp = *c;
        // *operator used to refer to the value
     *c = *d;
     *d = temp;
     printf("\n In function (Call By Reference
        Method) c = %d and d = %d", *c, *d);
  }
```

Output

```
In main(), a = 1 and b = 2
In function (Call By Value Method) a = 2 and
  b = 1
In main(), a = 1 and b = 2

In main(), c = 3 and d = 4
In function (Call By Reference Method) c = 4
  and d = 3
In main(), c = 4 and d = 3
```

3. Write a program to find biggest of three integers using functions.

```
#include <stdio.h>
int greater(int a, int b, int c);
int main()
{
   int num1, num2, num3, large;
   printf("\n Enter the first number: ");
   scanf("%d", &num1);
   printf("\n Enter the second number: ");
   scanf("%d", &num2);
   printf("\n Enter the third number: ");
   scanf("%d", &num3);
   large = greater(num1, num2, num3);
   printf("\n Largest number = %d", large);
   return 0;
}

int greater(int a, int b, int c)
{
   if(a>b && a>c)
      return a;
   if(b>a && b>c)
      return b;
   else
      return c;
}
```

Output

```
Enter the first number : 45
Enter the second number: 23
Enter the third number : 34
Largest number = 45
```

4. Write a program to calculate area of a circle using a function.

```
#include <stdio.h>
float cal_area(float r);

int main()
{
   float area, radius;
   printf("\n Enter the radius of the
      circle: ");
   scanf("%f", &radius);
   area = cal_area(radius);
   printf("\n Area of the circle with radius
      %f = %f", radius, area);
   return 0;
}
float cal_area(float radius)
{
   return (3.14 * radius * radius);
}
```

Output

```
Enter the radius of the circle: 7
Area of the circle with radius 7 = 153.83
```

5. Write a program, using functions, to find whether a number is even or odd.

```
#include <stdio.h>
int evenodd(int);
int main()
{
   int num, flag;
   printf("\n Enter the number: ");
```

```
    scanf("%d", &num);
    flag = evenodd(num);
    if (flag == 1)
      printf("\n %d is EVEN", num);
    else
      printf("\n %d is ODD", num);
    return 0;
}
int evenodd(int a)
{
   if(a%2 == 0)
     return 1;
   else
     return 0;
}
```

Output

```
Enter the number : 42
EVEN
```

6. Write a program to convert time to minutes.

```
#include <stdio.h>
#include <conio.h>
int convert_time_in_mins(int hrs, int
  minutes);
main()
{
   int hrs, minutes, total_mins;
   printf("\n Enter hours and minutes: ");
   scanf("%d %d", &hrs, &minutes);
   total_mins = convert_time_in_mins(hrs,
        minutes);
   printf("\n Total minutes = %d", total_
        mins);
   getch();
   return 0;
}

int convert_time_in_mins(int hrs, int minutes)
{
   int mins;
   mins = hrs * 60 + minutes;
   return mins;
}
```

Output

```
Enter the hours and minutes: 4 30
Total minutes = 270
```

7. Write a program to calculate P(*n*/*r*).

```
#include <stdio.h>
#include <conio.h>
int fact(int);
main()
{
   int n, r;
   float result;
   clrscr();
   printf("\n Enter the value of n: ");
   scanf("%d", &n);
   printf("\n Enter the value of r: ");
   scanf("%d", &r);
   result = (float)Fact(n)/Fact(r);
   printf("\n P(n/r): P(%d)/(%d) = %f", n,
        r, result);
   getch();
   return 0;
}
int Fact(int num)
{
   int f = 1, i;
   for(i=num; i>=1; i--)
      f = f * i;
   return f;
}
```

Output

```
Enter the value of n: 4
Enter the value of r: 2
P(n/r): P(12)/(3) = 12.00
```

8. Write a program to calculate C(*n*/*r*).

```
#include <stdio.h>
#include <conio.h>
int fact(int);
main()
{
   int n, r;
   float result;
   clrscr();
   printf("\n Enter the value of n: ");
   scanf("%d", &n);
   printf("\n Enter the value of r: ");
   scanf("%d", &r);
   result = (float)Fact(n)/(Fact(r)*Fact(n-r));
   printf("\n C(n/r) : C(%d/%d) = %.2f", n,
        r, result);
   getch();
   return 0;
}
int Fact(int num)
{
   int f = 1, i;
   for(i=num; i>=1; i--)
      f = f * i;
   return f;
}
```

Output

```
Enter the value of n: 4
Enter the value of r: 2
C(n/r): C(12)/(3) = 6.00
```

9. Write a program to sum the series 1/1! + 1/2! + 1/3! + ... + 1/*n*!

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

```
int fact(int);
main()
{
   int n, f, i;
   float result = 0.0;
   clrscr();
   printf("\n Enter the value of n: ");
   scanf("%d", &n);
   for(i=1; i<=n; i++)
   {
      f = Fact(i);
      result += 1/(float)f;
   }
   printf("\n The sum of the series 1/1! +
         1/2! + 1/3!... = %f", result);
   getch();
   return 0;
}
int Fact(int num)
{
   int f = 1, i;
   for(i=num; i>=1; i--)
      f = f * i;
   return f;
}
```

Output

```
Enter the value of n: 2
The sum of the series 1/1! + 1/2! + 1/3!... = 1.5
```

10. Write a program to sum the series 1/1! + 4/2! + 27/3! + ...

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int fact(int);
main()
{
   int n, i, NUM, DENO;
   float sum = 0.0;
   clrscr();
   printf("\n ");
   scanf("%d",&n);
   for(i=1; i<=n; i++)
   {
      NUM = pow(i,i);
      DENO = Fact(i);
      sum += (float)NUM/DENO;
   }
   printf("\n 1/1! + 4/2! + 27/3! + ....=
         %f", sum);
   getch();
   return 0;
}
int Fact(int n)
{
   int f = 1, i;
   for(i=n; i>=1; i--)
      f = f * i;
   return f;
}
```

Output

```
Enter the value of n: 2
1/1! + 4/2! + 27/3! + ....= 3
```

10.8 SCOPE OF VARIABLES

In C, all constants and variables have a defined scope. By scope we mean the accessibility and visibility of the variables at different points in the program. A variable or a constant in C has four types of scope: block, function, file, and program scope.

10.8.1 Block Scope

We have studied that a statement block is a group of statements enclosed within opening and closing curly brackets ({ }). If a variable is declared within a statement block then, as soon as the control exits that block, the variable will cease to exist. Such a variable, also known as a local variable, is said to have a *block scope*.

So far we had been using local variables. For example, if we declare an integer x inside a function, then that variable is unknown to the rest of the program (i.e., outside that function).

Programming Tip:
It is an error to use the name of a function argument as the name of a local variable.

Blocks of statements may be placed one after the other in a program; such blocks that are placed at the same level are known as *parallel blocks*. However, a program may also contain a nested block, such as a while loop inside `main()`. If an integer x is declared and initialized in the `main()`, and then re-declared and re-initialized in the `while` loop, then the integer variable x will occupy different memory slots and will be considered as different variables. The following code reveals this concept.

```
#include <stdio.h>
int main()
{
   int x = 10;
   int i = 0;
   printf("\n The value of x outside the
         while loop is %d", x);
   while (i<3)
   {
      int x = i;
      printf("\n The value of x inside the
         while loop is %d", x);
      i++;
   }
   printf("\n The value of x outside the
         while loop is %d", x);
   return 0;
}
```

Output

```
The value of x outside the while loop is 10
The value of x inside the while loop is 0
The value of x inside the while loop is 1
The value of x inside the while loop is 2
The value of x outside the while loop is 10
```

Programming Tip: Try to avoid variable names that hide variables in outer scopes.

You may get an error message while executing this code. This is because some C compilers make it mandatory to declare all the variables first, i.e., they permit declaration of variables right after the curly brackets of `main()` starts.

Hence, we can conclude two things:

- Variables declared with same names as those in outer blocks, mask the outer block variables while executing the inner block.
- In nested blocks, variables declared outside the inner blocks are accessible to the nested blocks, provided these variables are not re-declared within the inner block.

Note In order to avoid error, a programmer must prefer to use different names for the variables in different blocks.

10.8.2 Function Scope

Function scope indicates that a variable is active and visible from the beginning to the end of a function. In C, only the `goto` label has function scope. In other words function scope is applicable only with `goto` label names. This means that the programmer cannot have the same label name inside a function.

Using `goto` statements is not recommended as it is not considered to be a good programming practice. We will not discuss the `goto` statement in detail here, but we will take a look at an example of code that demonstrates the function scope.

```
int main()
{
   ...
   loop:   /* A goto label has function
   scope */
   ...
   goto loop;  /* the goto statement */
   ...
   return 0;
}
```

In this example, the label `loop` is visible from the beginning to the end of the `main()` function. Therefore, there should not be more than one label having the same name within the `main()` function.

10.8.3 Program Scope

Till now we have studied that variables declared within a function are local variables. These local variables (also known as internal variables) are automatically created when they are declared in the function and are usable only within that function. The local variables are unknown to other functions in the program. Such variables cease to exist after the last statement in the function in which they are declared and are re-created each time the function is called.

However, if you want this function to access some variables which are not passed to them as arguments, then declare those variables outside any function blocks. Such variables are commonly known as *global variables* and can be accessed from any point in the program.

Programming Tip: The value of a global variable can be used from anywhere in the program whereas the local variable cease to exist outside the function in which it is declared.

Lifetime Global variables are created at the beginning of program execution and remain in existence throughout the period of execution of the program. These variables are known to all the functions in the program and are accessible to them for usage. Global variables are not limited to a particular function so they exist even when a function calls another function. These variables retain their value so that they can be used from every function in the program.

Place of declaration The global variables are declared outside all the functions including `main()`. Although there is no specific rule that states where the global variables must be declared, it is always recommended to declare them at the beginning of the program code.

Name conflict If we have a variable declared in a function that has same name as that of the global variable, then the function will use the local variable declared within it and ignore the global variable. However, the programmer must not use names of global variables as the names of local variables, as this may lead to confusion and lead to erroneous result.

Note If a global variable is not initialized during its initialization then it is automatically initialized to zero by default.

Consider the following program.

```
#include <stdio.h>
int x = 10;
void print();
int main()
{
   printf("\n The value of x in main() =
          %d", x);
   int x = 2;
   printf("\n The value of local variable x
          in main() = %d", x);
   print();
   return 0;
}
void print()
```

```
{
    printf("\n The value of x in print() =
        %d", x);
}
```

Output

```
The value of x in main() = 10
The value of local variable x in main() = 2
The value of x in print() = 10
```

From the code we see that local variables overwrite the value of global variables. In big programs use of global variables is not recommended until it is very important to use them because there is a big risk of confusing them with any local variables of the same name.

Note: Functions are considered to be self-contained and independent modules. So they need to be isolated from the rest of the code, but using global variables negates this idea behind making functions.

10.8.4 File Scope

When a global variable is accessible until the end of the file, the variable is said to have file scope. To allow a variable to have file scope, declare that variable with the static keyword before specifying its data type, as shown:

```
static int x = 10;
```

A global static variable can be used anywhere from the file in which it is declared but it is not accessible by any other files. Such variables are useful when the programmer writes his/her own header files.

10.9 STORAGE CLASSES

The storage class of a variable defines the scope (visibility) and lifetime of variables and/or functions declared within a C program. In addition to this, the storage class gives the following information about the variable or the function.

- The storage class of a function or a variable determines the part of memory where storage space will be allocated for that variable or function (whether the variable/function will be stored in a register or in RAM).
- It specifies how long the storage allocation will continue to exist for that function or variable.
- It specifies the scope of the variable or function, i.e., the storage class indicates the part of the C program in which the variable name is visible or the part in which it is accessible. In other words, whether the variable/function can be referenced throughout the program or only within the function, block, or source file where it has been defined.
- It specifies whether the variable or function has internal, external, or no linkage.
- It specifies whether the variable will be automatically initialized to zero or to any indeterminate value.

C supports four storage classes: automatic, register, external, and static. The general syntax for specifying the storage class of a variable can be given as

```
<storage_class_specifier> <data type>
    <variable name>
```

10.9.1 Auto

The auto storage class specifier is used to explicitly declare a variable with *automatic storage*. It is the default storage class for variables declared inside a block. For example, if we write

```
auto int x;
```

then x is an integer that has automatic storage. It is deleted when the block in which x was declared exits.

The auto storage class can be used to declare variables in a block or the names of function parameters. However, since the variable names or names of function parameters have automatic storage by default, the auto storage class specifier is therefore treated as redundant while declaring data.

Important things to remember about variables declared with auto storage class include:

- All local variables declared within a function belong to automatic storage class by default.
- They should be declared at the start of the program block (right after the opening curly bracket {).
- Memory for the variable is automatically allocated upon entry to a block and freed automatically upon exit from that block.
- The scope of the variable is local to the block in which it is declared. These variables may be declared within a nested block.
- Every time the block (in which the automatic variable is declared) is entered, the value of the variable declared with initializers is initialized.
- The auto variables are stored in the primary memory of the computer.
- If auto variables are not initialized at the time of declaration, then they contain some garbage value.

The following code uses an auto integer that is local to the function in which it is defined.

```
#include <stdio.h>
void func1()
{
    int a = 10;
    printf("\n a = %d", a);
        // auto integer local to func1()
}
void func2()
{
    int a = 20;
    printf("\n a = %d", a);
        // auto integer local to func2()
}
```

```
void main()
{
   int a = 30  // auto integer local to main()
   func1();
   func2();
   printf("\n a = %d", a);
}
```

Output

```
a = 10
a = 20
a = 30
```

10.9.2 Register

When a variable is declared using `register` as its storage class, it is stored in a CPU register instead of RAM. Since the variable is stored in RAM, the maximum size of the variable is equal to the register size. One drawback of using a register variable is that they cannot be operated using the unary '&' operator because it does not have a memory location associated with it. A register variable is declared in the following manner:

```
register int x;
```

Register variables are used when quick access to the variable is needed. It is not always necessary that the register variable will be stored in the register. Rather, the register variable might be stored in a register depending on the hardware and implementation restrictions.

Hence, programmers can only suggest the compiler to store those variables in the registers which are used repeatedly or whose access times are critical. However, for the compiler, it is not an obligation to always accept such requests. In case the compiler rejects the request to store the variable in the register, the variable is treated as having the storage class specifier `auto`.

Similar to `auto` variables, `register` variables also have automatic storage duration. That is, each time a block is entered, the storage for register variables defined in that block is accessible and the moment that block is exited, the variables becomes no longer accessible for use. Now let us have a look at the following code that uses a `register` variable.

```
#include <stdio.h>
int exp(int a, int b); // FUNCTION DECLARATION
main()
{
   int a = 3, b = 5, res;
   res = exp(a, b);// FUNCTION CALL
   printf("\n %d to the power of %d = %d",
         a, b, res);
   getch();
   return 0;
}
int exp(int a, int b) // FUNCTION DEFINITION
{
   register int res=1;
   int i;
   for(i=1; i<=b; i++)
      res = res * b;
   return res;
}
```

Output

```
3 to the power of 5 = 243
```

10.9.3 Extern

A large C program can be broken down into smaller programs. When these smaller programs are compiled, they are joined together to form a large program. However, these smaller programs may need to share certain variables for processing. For such situations C language provides an external storage class that is specified using the keyword `extern`.

The `extern` storage class is used to give a reference of a global variable that is visible to all the program files. Such global variables are declared like any other variable in one of the program file. When there are multiple files in a program and you need to use a particular function or variable in a file apart from which it is declared, then use the keyword `extern`. To declare a variable `x` as `extern` write,

```
extern int x;
```

External variables may be declared outside any function in a source code file as any other variable is declared. But usually external variables are declared and defined at the beginning of a source file.

Memory is allocated for external variables when the program begins execution, and remains allocated until the program terminates. External variables may be initialized while they are declared. However, the initializer must be a constant expression. The compiler will initialize its value only once during the compiler time. In case the `extern` variable is not initialized, it will be initialized to zero by default.

External variables have global scope, i.e., these variables are visible and accessible from all the functions in the program. However, if any function has a local variable with the same name and types as that of the global or extern variable, then references to the name will access the local variable rather than the extern variable. Hence, extern variables are overwritten by local variables.

Let us now write a program in which we will use the `extern` keyword.

```
// FILE 1.C

#include <stdio.h>
#include <FILE2.C>
```

```
// Programmer's own header file
int x;
void print(void);
int main()
{
   x = 10;
   printf("\n x in FILE1 = %d", x);
   print();
   return 0;
}
// END OF FILE1.C
// FILE2.C
#include <stdio.h>
extern int x;
void print()
{
   printf(«\n x in FILE 2 = %d», x);
}
main()
{
// Statements
}
// END OF FILE2.C
```

Output

```
x in FILE1 = 10
x in FILE2 = 10
```

In the program, we have used two files—`File1` and `File2`. `File1` has declared a global variable `x`. `File1` also includes `File2` which has a print function that uses the external variable `x` to print its value on the screen.

Note The extern specifier tells the compiler that the variable has already been declared elsewhere and therefore it should not allocate storage space for that variable again. During compilation of the program, the linker will automatically resolve the reference problem.

In a multi-file program, the global variable must be declared only once (in any one of the files) without using the extern keyword. This is because, otherwise the linker will have a conflict as to which variable to use and therefore in such a situation it raises a warning message.

10.9.4 Static

While `auto` is the default storage class for all local variables, `static` is the default storage class for all global variables. Static variables have a lifetime over the entire program, i.e., memory for the `static` variables is allocated when the program begins running and is freed when the program terminates. To declare an integer `x` as `static`, write,

```
static int x = 10;
```

Here `x` is a local static variable. Static local variables when defined within a function are initialized at the runtime. The difference between an `auto` variable and a static variable is that the static variable when defined within a function is not re-initialized when the function is called again and again. It is initialized just once and further calls of the function share the value of the static variable. Hence, the static variable inside a function retains its value during various calls.

When a `static` variable is not explicitly initialized by the programmer then it is automatically initialized to zero when memory is allocated for them. Although `static` automatic variables exist even after the block in which they are defined terminates, their scope is local to the block in which they are defined.

`Static` storage class can be specified for `auto` as well as `extern` variables. For example we can write,

```
static extern int x;
```

When we declare a variable as an `external static` variable, then that variable is accessible from all the functions in this source file.

Note Static variables can be initialized while they are being declared. But this initialization is done only once at the compile time when memory is being allocated for the static variable. Also the value with which the static variable is initialized must be a constant expression.

Look at the following code which clearly differentiates between a static variable and a normal variable.

```
#include <stdio.h>
void print(void);  // FUNCTION DECLARATION
int main()
{
   printf("\n First call of print()");
   print();       // FUNCTION CALL
   printf("\n\n Second call of print()");
   print();       // FUNCTION CALL
   printf("\n\n Third call of print()");
   print();       // FUNCTION CALL
   return 0;
}

void print()  // FUNCTION DEFINITION
{
   static int x;
   int y = 0;
   printf("\n Static integer variable, x =
         %d", x);
   printf("\n Integer variable, y = %d", y);
   x++;
   y++;
}
```

Output

```
First call of print()
Static integer variable, x = 0
Integer variable, y = 0
```

Table 10.2 Comparison of storage classes

Feature	Storage Class			
	auto	extern	register	static
Accessibility	Accessible within the function or block in which it is declared.	Accessible within all program files that are a part of the program.	Accessible within the function or block in which it is declared.	Local: Accessible within the function or block in which it is declared. Global: Accessible within the program in which it is declared.
Storage	Main memory	Main memory	CPU register	Main memory
Existence	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared.	Exists throughout the execution of the program.	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared.	Local: Retains value between function calls or block entries. Global: Preserves value in program files.
Default value	Garbage	Zero	Garbage	Zero

```
Second call of print()
Static integer variable, x = 1
Integer variable, y = 0

Third call of print()
Static integer variable, x = 2
Integer variable, y = 0
```

10.9.5 Comparison of Storage Classes

Table 10.2 compares the key features of all the storage classes.

10.10 RECURSIVE FUNCTIONS

A *recursive function* is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Every recursive solution has two major cases, they are

- *Base case,* in which the problem is simple enough to be solved directly without making any further calls to the same function.
- *Recursive case,* in which first the problem at hand is divided into simpler sub parts. Second the function calls itself but with sub parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Therefore, recursion is defining large and complex problems in terms of a smaller and more easily solvable problem. In recursive function, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.

To understand recursive functions, let us take an example of calculating factorial of a number. To calculate `n!`, we have to multiply the number with factorial of the number that is 1 less than that number. In other words, `n! = n x (n-1)!`

Let us say we need to find the value of 5!

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

This can be written as

$5! = 5 \times 4!$, where $4! = 4 \times 3!$

Therefore,

$5! = 5 \times 4 \times 3!$

Similarly, we can also write,

$5! = 5 \times 4 \times 3 \times 2!$

Expanding further

$5! = 5 \times 4 \times 3 \times 2 \times 1!$

We know, $1! = 1$

Therefore, the series of problem and solution can be given as shown in Figure 10.6.

Problem	Solution
$5!$	$5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4!$	$= 5 \times 4 \times 3 \times 2 \times 1$
$= 5 \times 4 \times 3!$	$= 5 \times 4 \times 3 \times 2$
$= 5 \times 4 \times 3 \times 2!$	$= 5 \times 4 \times 6$
$= 5 \times 4 \times 3 \times 2 \times 1!$	$= 5 \times 24$
	$= 120$

Figure 10.6 Recursive factorial function

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number. Every recursive function must have a base case and a recursive case. For the factorial function,

Programming Tip: Every recursive function must have at least one base case. Otherwise, the recursive function will generate an infinite sequence of calls thereby resulting in an error condition known as an infinite stack.

- *Base case* is when $n = 1$, because if $n = 1$, the result will be 1 as $1! = 1$.
- *Recursive case* of the factorial function will call itself but with a smaller value of n, this case can be given as

$$\text{factorial}(n) = n \times \text{factorial}(n-1)$$

Look at the following code which calculates the factorial of a number recursively.

```
#include <stdio.h>
int Fact(int);  // FUNCTION DECLARATION
main()
{
    int num;
    printf("\n Enter the number: ");
    scanf("%d", &num);
    printf("\n Factorial of %d = %d", num,
      Fact(num));// FUNCTION CALL
    return 0;
}

int Fact( int n) // FUNCTION DEFINITION
{
  if(n==1)
    return 1;
  return (n * Fact(n-1)); // FUNCTION CALL
}
```

From this example, let us analyse the basic steps of a recursive program.

Step 1: Specify the base case which will stop the function from making a call to itself.

Step 2: Check to see whether the current value being processed matches with the value of the base case. If yes, process and return the value.

Step 3: Divide the problem into a smaller or simpler sub-problem.

Step 4: Call the function from the sub-problem.

Step 5: Combine the results of the sub-problems.

Step 6: Return the result of the entire problem.

Note: The base case of a recursive function acts as the terminating condition. So, in the absence of an explicitly-defined base case, a recursive function would call itself indefinitely.

10.10.1 Greatest Common Divisor

The greatest common divisor (GCD) of two numbers (integers) is the largest integer that divides both the numbers. We can find the GCD of two numbers recursively by using *Euclid's algorithm* which states that

$$\text{GCD(a, b)} = \begin{cases} \text{b, if b divides a} \\ \text{GCD(b, a mod b), otherwise} \end{cases}$$

GCD can be implemented as a recursive function because if b does not divide a, then we call the same function (GCD) with another set of parameters that are smaller and simpler than the original ones. Here we assume that a > b. However if a < b, then interchange a and b in the formula.

Working

```
Assume a = 62 and b = 8
GCD(62, 8)
   rem = 62 % 8 = 6
   GCD(8, 6)
      rem = 8 % 6 = 2
      GCD(6, 2)
         rem = 6 % 2 = 0
      Return 2
   Return 2
Return 2
```

11. Write a program to calculate GCD using recursive functions.

```
#include <stdio.h>
int GCD(int, int);  // FUNCTION DECLARATION
main()
{
   int num1, num2, res;
   printf("\n Enter the two numbers: ");
   scanf("%d %d", &num1, &num2);
   res = GCD(num1, num2); // FUNCTION CALL
   printf("\n GCD of %d and %d = %d", num1,
        num2, res);
   return 0;
}
int GCD(int x, int y) // FUNCTION DEFINITION
{
   int rem;
   rem = x%y;
   if(rem==0)
      return y;
   else
      return (GCD(y, rem)); // FUNCTION CALL
}
```

10.10.2 Finding Exponents

We can find a solution to find exponent of a number using recursion. To find x^y, the base case would be when y = 0, as we know that any number raise to the power 0 is 1. Therefore, the general formula to find x^y can be given as

$$EXP(x, y) = \begin{cases} 1, & \text{if } y == 0 \\ x * EXP(x^{y-1}), & \text{otherwise} \end{cases}$$

Working

```
exp_rec(2, 4) = 2 * exp_rec(2, 3)
    exp_rec(2, 3) = 2 * exp_rec(2, 2)
      exp_rec(2, 2) = 2 * exp_rec(2, 1)
        exp_rec(2, 1) = 2 * exp_rec(2, 0)
          exp_rec(2, 0) = 1
        exp_rec(2, 1) = 2 * 1 = 2
      exp_rec(2, 2) = 2 * 2 = 4
  exp_rec(2, 3) = 2 * 4 = 8
exp_rec(2, 4) = 2 * 8 = 16
```

12. Write a program to calculate exp(x,y) using recursive functions.

```
#include <stdio.h>
int exp_rec(int, int);
main()
{
   int num1, num2, res;
   printf("\n Enter the two numbers: ");
   scanf("%d %d", &num1, &num2);
   res = exp_rec(num1, num2);
   printf("\n Result = %d", res);
   return 0;
}
int exp_rec(int x, int y)
{
   if(y==0)
      return 1;
   else
      return (x * exp_rec(x, y-1));
}
```

10.10.3 The Fibonacci Series

The Fibonacci series can be given as

0 1 1 2 3 5 8 13 21 34 55 ...

That is, the third term of the series is the sum of the first and second terms. On similar grounds, fourth term is the sum of second and third terms, and so on. Now we will design a recursive solution to find the n^{th} term of the Fibonacci series. The general formula to do so can be given as

$$FIB(n) = \begin{cases} 1, \text{ if } n <= 2 \\ FIB(n-1) + FIB(n-2), \text{ otherwise} \end{cases}$$

As per the formula, FIB(1) = 1 and FIB(2) = 1. So we have two base cases. This is necessary because every problem is divided into two smaller problems Figure 10.7).

13. Write a program to print the Fibonacci series using recursion.

```
#include <stdio.h>
int Fibonacci(int);
main()
{
   int n;
   printf("\n Enter the number of terms in
        the series: ");
   scanf("%d", &n);
   for(int i=0; i<n; i++)
      printf("\n Fibonacci (%d) = %d", i,
         Fibonacci(i));
   return 0;
}

int Fibonacci(int num)
{
   if(num <= 2)
      return 1;
   return (Fibonacci (num - 1) +
      Fibonacci(num - 2));
}
```

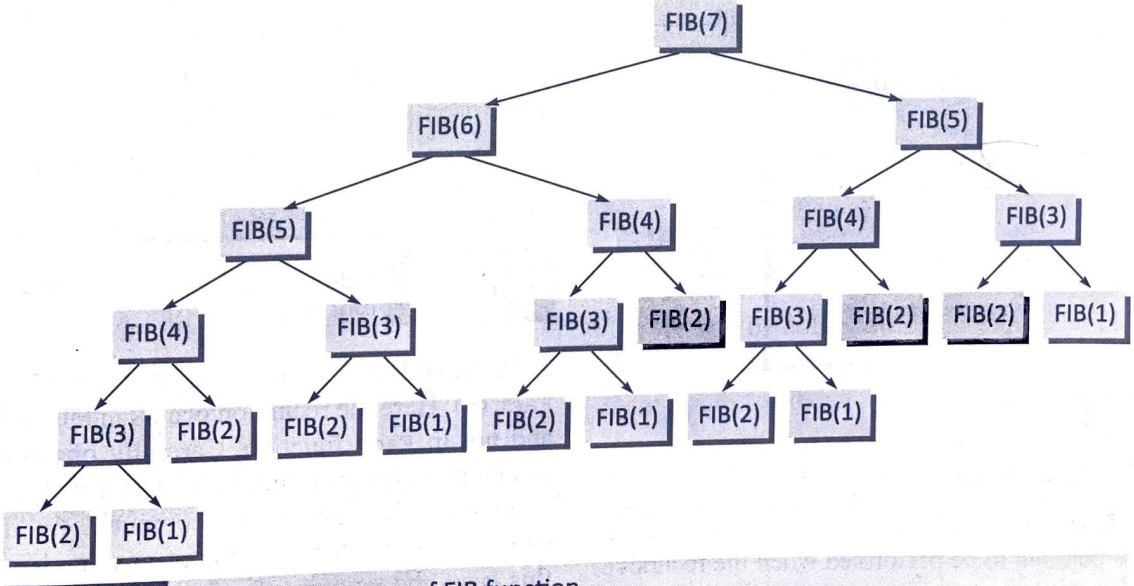


Figure 10.7 Recursion structure of FIB function

10.11 TYPES OF RECURSION

Recursion is a technique that breaks a problem into one or more sub-problems that are similar to the original problem. Any recursive function can be characterized based on the following:

- Whether the function calls itself directly or indirectly (*direct or indirect recursion*)
- Whether any operation is pending at each recursive call (*tail-recursive* or not)
- The structure of the calling pattern (*linear or tree-recursive*)

In this section, we will read about all these types of recursions.

10.11.1 Direct Recursion

A function is said to be *directly* recursive if it explicitly calls itself. For example, consider Figure 10.8.

```
int Func(int n)
{
    if(n==0)
        return n;
    return(Func(n-1));
}
```

Figure 10.8 Direct recursion

Here, the function `Func()` calls itself for all positive values of `n`, so it is said to be a directly recursive function.

10.11.2 Indirect Recursion

A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it. Look at the functions given in Figure 10.9. These two functions are indirectly recursive as they both call each other.

```
int Func1(int n)
{
    if(n==0)
        return n;
    return Func2(n);
}

int Func2(int x)
{
    return Func1(x-1);
}
```

Figure 10.9 Indirect recursion

10.11.3 Tail Recursion

A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller (Figure 10.10). When the called function returns, the returned value is immediately returned from the calling function. Tail recursive functions are highly desirable because they are much more efficient to use as the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

```
int Fact(int n)
{
    if(n==1)
        return 1;
    return (n * Fact(n-1));
}
```

Figure 10.10 Tail recursion

For example, the factorial function that we have written is a non-tail-recursive function, because there is a pending operation of multiplication to be performed on return from each recursive call.

Whenever there is a pending operation to be performed, the function becomes non-tail recursive. In such a non-tail recursive function, information about each pending operation must be stored, so the amount of information directly depends on the number of calls.

However, the same factorial function can be written in a tail-recursive manner as shown in Figure 10.11.

```
int Fact(n)
{
    return Fact1(n, 1);
}

int Fact1(int n, int res)
{
    if (n==1)
        return res;
    return Fact1(n-1, n*res);
}
```

Figure 10.11 Tail-recursive and non-tail-recursive functions

In the code, `Fact1` function preserves the syntax of the `Fact(n)`. Here the recursion occurs in the `Fact1` function and not in `Fact` function. Carefully observe that `Fact1` has no pending operation to be performed on return from recursive calls. The value computed by the recursive call is simply returned without any modification. So in this case, the amount of information to be stored on the

```
int Fibonacci(int num)
{
  if(num <= 2)
    return 1;
  return (Fibonacci (num - 1) + Fibonacci(num - 2));
}
```

Observe the series of function calls. When the function returns, the pending operations in turn calls the function:

Fibonacci(7) = Fibonacci(6) + Fibonacci(6) = Fibonacci(5) + Fibonacci(4)
Fibonacci(5) = Fibonacci(4) + Fibonacci(3)
Fibonacci(4) = Fibonacci(3) + Fibonacci(2)
Fibonacci(3) = Fibonacci(2) + Fibonacci(1)

Now we have, Fibonacci(3) = 1 + 1 = 2
Fibonacci(4) = 2 + 1 = 3
Fibonacci(5) = 3 + 2 = 5
Fibonacci(6) = 3 + 5 = 8
Fibonacci(7) = 5 + 8 = 13

Figure 10.12 Tree recursion

system stack is constant (just the value of `n` and `res` needs to be stored) and is independent of the number of recursive calls.

Converting Recursive Functions to Tail Recursive

A non-tail recursive function can be converted into a tail-recursive function by using an *auxiliary parameter* as we did in case of the Factorial function. The auxiliary parameter is used to form the result. When we use such a parameter, the pending operation is incorporated into the auxiliary parameter so that the recursive call no longer has a pending operation. We generally use an auxiliary function while using the auxiliary parameter. This is done to keep the syntax clean and to hide the fact that auxiliary parameters are needed.

10.11.4 Linear and Tree Recursion

Recursive functions can also be characterized depending on the way in which the recursion grows in a linear fashion or forming a tree structure (Figure 10.12).

In simple words, a recursive function is said to be *linearly* recursive when the pending operation (if any) does not make another recursive call to the function. For example, observe the last line of recursive factorial function. The factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another recursive call to `Fact`.

On the contrary, a recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function. For example, the `Fibonacci` function FIB in which the pending operations recursively calls the FIB function.

10.12 TOWER OF HANOI

The tower of Hanoi is one of the main applications of a recursion. It says, 'if you can solve $n - 1$ cases, then you can easily solve the n^{th} case'.

Look at Figure 10.13 which shows three rings mounted on pole A. The problem is to move all these rings from pole A to pole C while maintaining the same order. The main issue is that the smaller disk must always come above the larger disk.

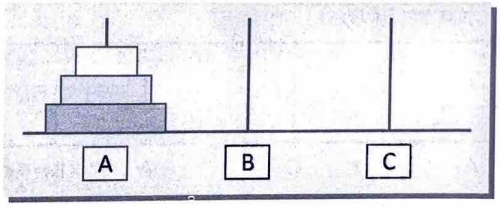


Figure 10.13 Tower of Hanoi

We will be doing this using a spare pole. In our case, A is the source pole, C is the destination pole, and B is the spare pole. To transfer all the three rings from A to C, we will first shift the upper two rings ($n - 1$ rings) from the source pole to the spare pole. We move the first two rings from pole A to B as shown in Figure 10.14.

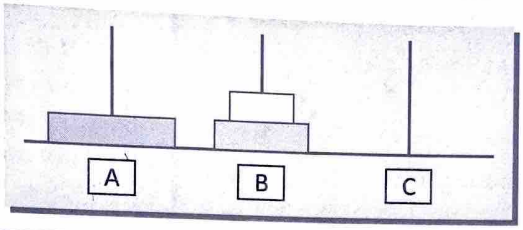


Figure 10.14 Move rings from A to B

Now that $n - 1$ rings have been removed from pole A, the source pole, the n^{th} ring can be easily moved from the source pole (A) to the destination pole (C). Figure 10.15 shows this step.

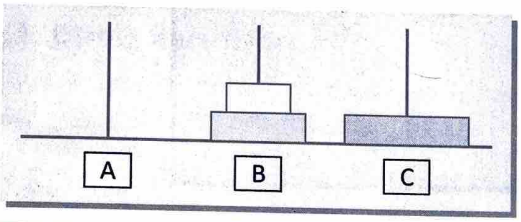


Figure 10.15 Move ring from A to C

The final step is to move the $n - 1$ rings from the spare pole (B) to the destination pole (C). This is shown in Figure 10.16.

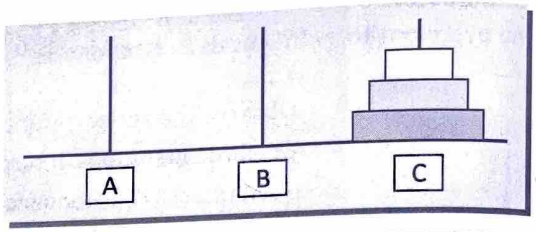


Figure 10.16 Move rings from B to C

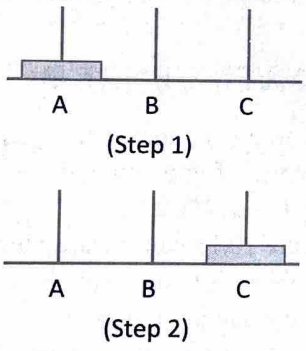
To summarize, the solution to our problem of moving n rings from A to C using B as spare can be given as:

Base case If $n = 1$

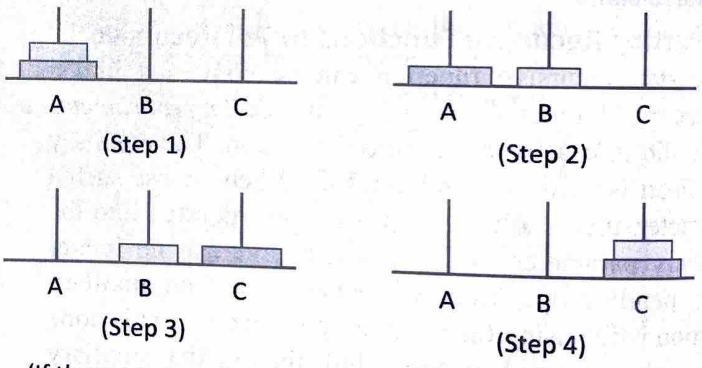
- Move the ring from A to C using B as spare

Recursive case

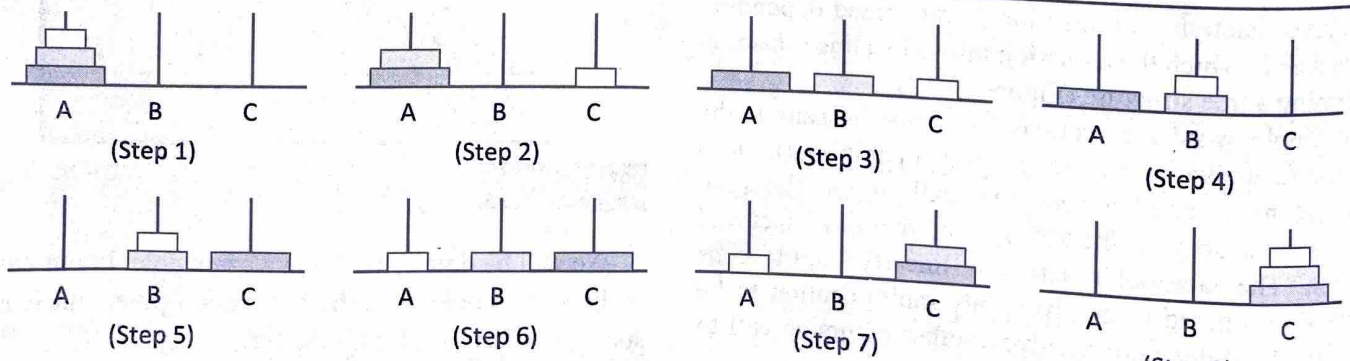
- Move $n - 1$ rings from A to B using C as spare
- Move the one ring left on A to C using B as spare
- Move $n - 1$ rings from B to C using A as spare



(If there is only one ring, then simply move the ring from source to the destination.)



(If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from the source to the destination.)



(Consider the working with three rings.)

Figure 10.17 Working of Tower of Hanoi with one, two, and three rings

The following code implements the solution of the Tower of Hanoi problem.

```
#include <stdio.h>
int main()
{
   int n;
   printf("\n Enter the number of rings: ");
   scanf("%d", &n);
   move(n,'A', 'C', 'B');
   return 0;
}
void move(int n, char source, char dest, char
   spare)
{
   if (n==1)
      printf("\n Move from %c to
          %c",source,dest);
   else
   {
      move(n-1,source,spare,dest);
      move(1,source,dest,spare);
      move(n-1,spare,dest,source);
   }
}
```

Let us look at the Tower of Hanoi problem in detail using the program given above. Figure 10.17 on the next page explains the working of the program using one, then two, and finally three rings.

10.13 RECURSION VERSUS ITERATION

Recursion is more of a top-down approach to problem solving in which the original problem is divided into smaller sub-problems. On the contrary, iteration follows a bottom-up approach that begins with what is known and then constructing the solution step by step.

Recursion is an excellent way of solving complex problems especially when the problem can be defined in recursive terms. For such problems, a recursive code can be written and modified in a much simpler and clearer manner.

However, recursive solutions are not always the best solutions. In some cases, recursive programs may require substantial amount of run-time overhead. Therefore, when implementing a recursive solution, there is a trade-off involved between the time spent in constructing and maintaining the program and the cost incurred in running-time and memory space required for the execution of the program.

Whenever a recursive function is called, some amount of overhead in the form of a run time stack is always involved. Before jumping to the function with a smaller parameter, the original parameters, the local variables, and the return address of the calling function are all stored on the system stack. Therefore, while using recursion a lot of time is needed to first push all the information on the stack when the function is called and then time is again involved in retrieving the information stored on the stack once the control passes back to the calling function.

To conclude, one must use recursion only to find solution to a problem for which no obvious iterative solution is known. To summarize the concept of recursion let us briefly discuss the pros and cons of recursion.

The advantages of using a recursive program include the following:

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion works similar to the original formula to solve a problem.
- Follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

The drawbacks/disadvantages of using a recursive program include the following:

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive process in midstream is slow and sometimes nasty.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counter part.
- It is difficult to find bugs, particularly while using global variables.

The advantages of recursion pays off for the extra overhead involved in terms of time and space required.

SUMMARY

- Every function in the program is supposed to perform a well defined task. The moment the compiler encounters a function call, the control jumps to the statements that are a part of the called function. After the called function is executed, the control is returned back to the calling program.
- All the libraries in C contain a set of functions that have been prewritten and pre-tested, so the programmers use them without worrying about the code details. This speeds up program development.
- While function declaration statement identifies a function with its name, a list of arguments that

SUMMARY

it accepts and the type of data it returns, the function definition, on the other hand, consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

- `main()` is the function that is called by the operating system and therefore, it is supposed to return the result of its processing to the operating system.
- Placing the function declaration statement prior to its use enables the compiler to make a check on the arguments used while calling that function.
- A function having void as its return type cannot return any value. Similarly, a function having void as its parameter list cannot accept any value.
- When a function is defined, space is allocated for that function in the memory. A function definition comprises of two parts: function header and function body.
- Call-by-value method passes values of the variables to the called function. Therefore, the called function uses a copy of the actual arguments to perform its intended task. This method is used when the function does not need to modify the values of the original variables in the calling program.
- In call-by-reference method, address of the variables are passed by the calling function to the called function. Hence, in this method, a function receives an implicit reference to the argument, rather than a copy of its value. This allows the function to modify the value of the variable and that change will be reflected in the calling function as well.
- Scope means the accessibility and visibility of the variables at different points in the program. A variable or a constant in C has four types of scope: block, function, file, and program scope.
- The storage class of a variable defines the scope (visibility) and life time of variables and/or functions declared within a C program.
- The auto storage class is the default storage class for variables declared inside a block. The scope of the variable is local to the block in which it is declared. When a variable is declared using register as its storage class, it is stored in a CPU register instead of RAM. Extern is used to give a reference of a global variable that is visible to all the program files. Static is the default storage class for global variables.
- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. They are implemented using system stack.

GLOSSARY

Argument A value passed to the called function by the calling function

Block A sequence of definitions, declarations, and statements, enclosed within curly brackets.

Divide and conquer Solving a problem by dividing it into two or more smaller instances. Each of these smaller instances is recursively solved, and the solutions are combined to produce a solution for the original problem.

Iteration Solving a problem by repeatedly working on successive parts of the problem.

Recursion An algorithmic technique where a function calls itself with a smaller of the task in order to solve that task.

Recursion termination The point at which base condition is met and a recursive algorithm stops calling itself and begins to return values.

Recursion tree Technique to analyse the complexity of an algorithm by diagramming the recursive function calls.

Tail recursion A form of recursion in which the last operation of a function is a recursive call.

Tower of Hanoi Given three towers or poles and *n* disks of decreasing sizes, move the disks from one pole to another one by one without putting a larger disk on a smaller one.