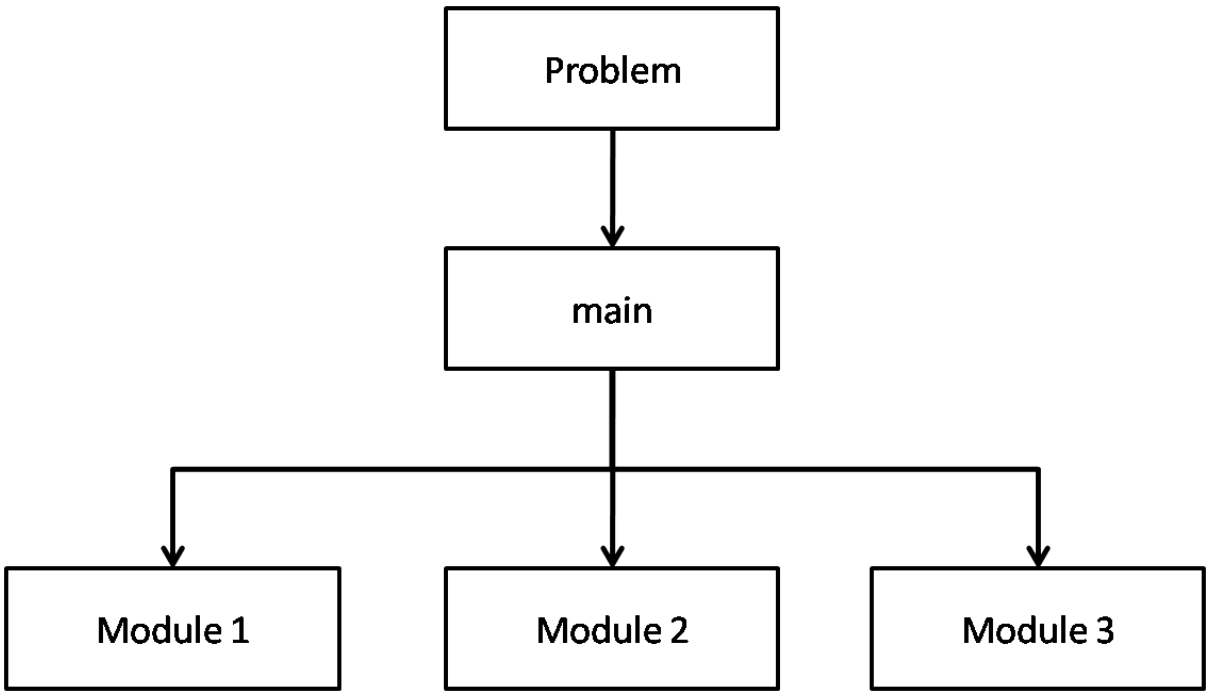
UNIT - 3

FUNCTIONS

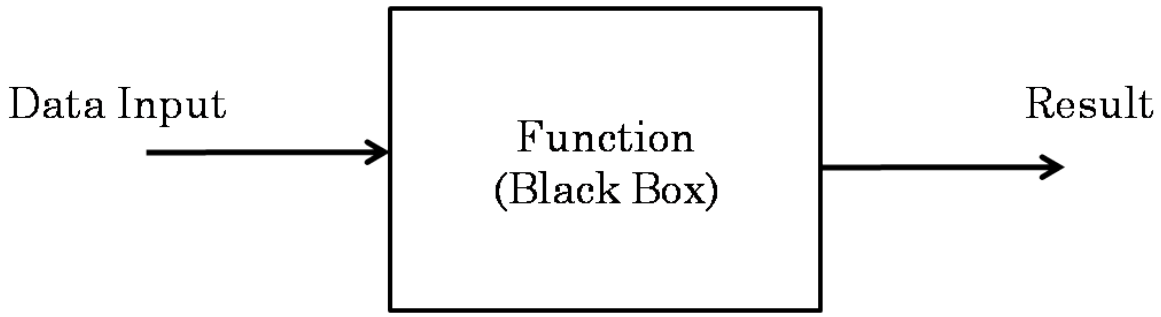
Functions

Until now, in all the C programs that we have written, the program consists of a **main** function and inside that we are writing the logic of the program. The disadvantage of this method is, if the logic/code in the **main** function becomes huge or complex, it will become difficult to debug the program or test the program or maintain the program. So, generally while writing the programs, the entire logic is divided into smaller parts and each part is treated as a function. This type of approach for solving the given problems is known as **Top Down** approach. The top-down approach of solving the given problem can be seen below:



Definition: A function is a self contained block of code that performs a certain task/job. For example, we can write a function for reading an integer or we can write a function to add numbers from 1 to 100 etc.

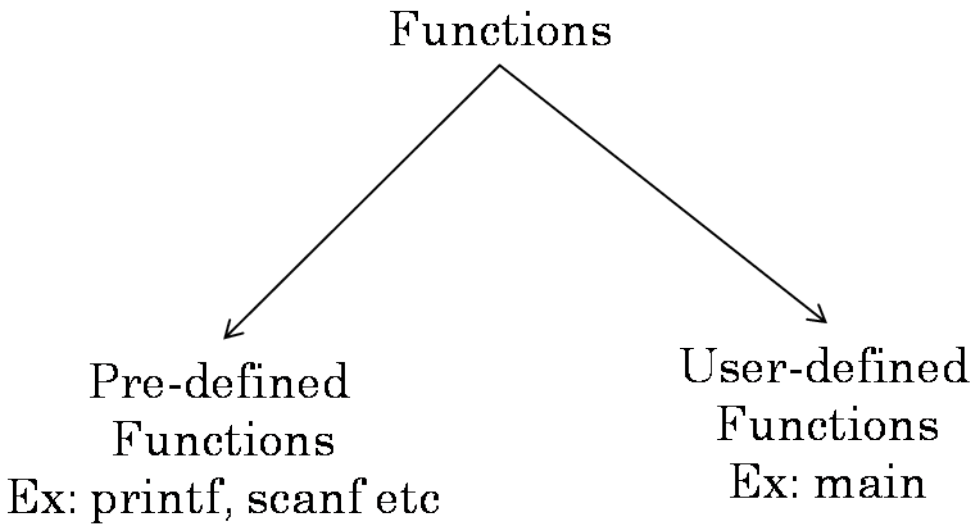
Generally a function can be imagined like a **Black Box**, which accepts data input and transforms the data into output results. The user knows only about the inputs and outputs. User has no knowledge of the process going on inside the box which converts the given input into output. This can be seen diagrammatically as shown below:



Types of Functions

Based on the nature of functions, they can be divided into two categories. They are:

1. Predefined functions / Library functions
2. User defined functions



Predefined functions / Library functions

A predefined function or library function is a function which is already written by another developer. The users generally use these library functions in their own programs for performing the desired task. The predefined functions are available in the header files. So, the user has to include the respective header file to use the predefined functions available in it.

For example, the **printf** function which is available in the **stdio.h** header file is used for printing information onto the console. Other examples of predefined functions are: scanf, gets, puts, getchar, putchar, strlen, strcat etc.

User defined functions

A user defined function is a function which is declared and defined by the user himself. While writing programs, if there are no available library functions for performing a particular task, we write our own function to perform that task. Example for user defined function is **main** function.

Need for functions

Functions have many advantages in programming. Almost all the languages support the concept of functions in some way. Some of the advantages of writing/using functions are:

1. Functions support top-down modular programming.
2. By using functions, the length of the source code decreases.
3. Writing functions makes it easier to isolate and debug the errors.
4. Functions allow us to reuse the code.

Functions Terminology

Creating functions

For creating functions in C programs, we have to perform two steps. They are: 1) Declaring the function and 2) Defining the function.

Declaring functions

The function declaration is the blue print of the function. The function declaration can also be called as the function's prototype. The function declaration tells the compiler and the user about what is the function's name, inputs and output(s) of the function and the return type of the function. The syntax for declaring a function is shown below:

```
return-type function-name(parameters list);
```

Example:

```
int readint( );
```

In the above example, **readint** is the name of the function, **int** is the return type of the function. In our example, **readint** function has no parameters. The parameters list is optional. The functions are generally declared in the global declaration section of the program.

Defining functions

The function definition specifies how the function will be working i.e the logic of the function will be specified in this step. The syntax of function definition is shown below:

```
return-type func-name(parameters list…)

{

        local variable declarations;

        -----

        -----

        return(expression);

}
```

In the above syntax the **return-type** can be any valid data type in C like: int, float, double, char etc. The **func-name** is the name of the function and it can be any valid identifier. The **parameters list** is optional. The **local variables** are the variables which belong to the function only. They are used within the body of the function, not outside the function. The **return** is a keyword in C. It is an unconditional branch statement. Generally, the **return** statement is used to return a value.

Example:

```
int readint( )
{
        int num;
        printf("Enter a number: ");
        scanf("%d",&num);
        return num;
}
```

In the above example, **readint** is the name of the function, **int** is the return type. The identifier **num** is a local variable with respect to **readint** function. The above function is reading an integer from the keyboard and returning that value back with the help of the **return** statement.

Note: *Care must be taken that the return type of the function and the data type of the value returned by the return statement must match with one another.*

Using Functions

After declaring and defining the functions, we can use the functions in our program. For using the functions, we must **call** the function by its name. The syntax for calling a function is as shown below:

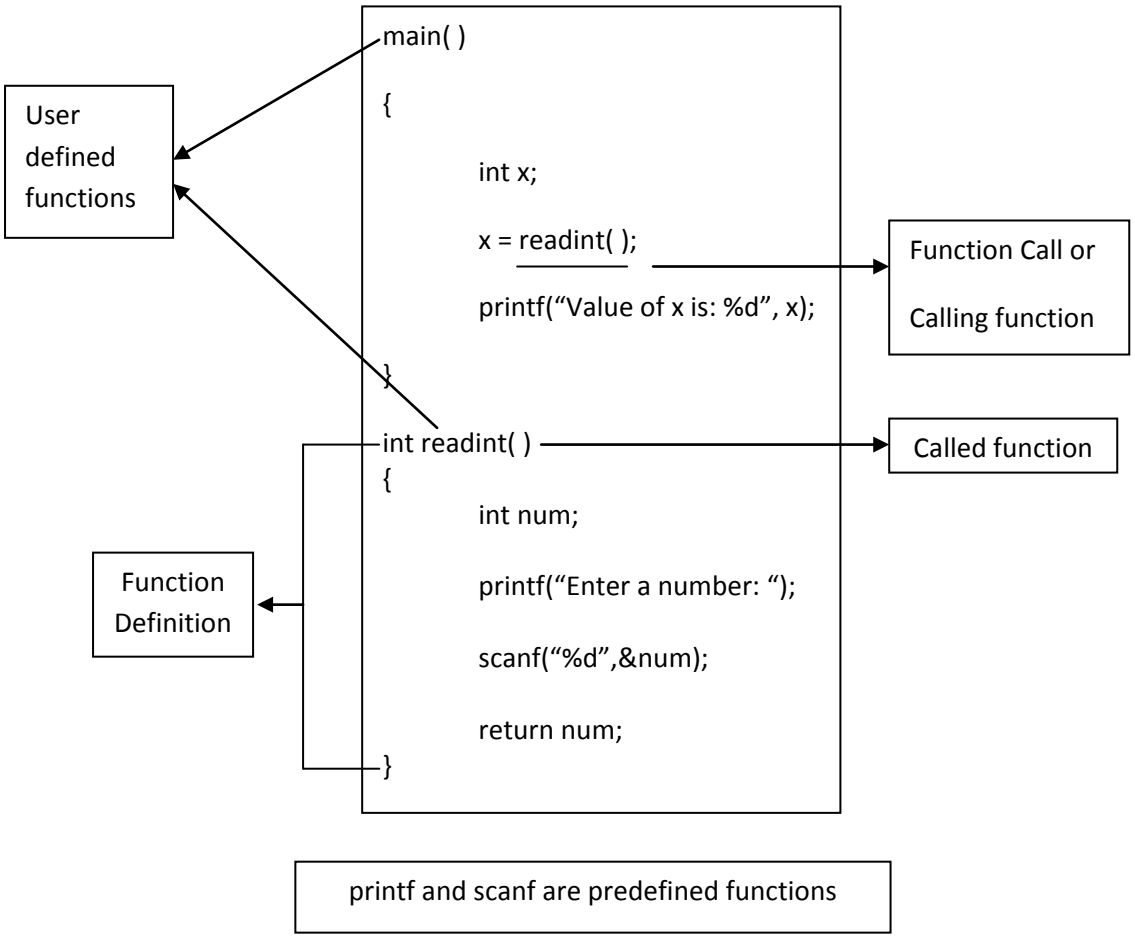
```
function-name(parameters list);
```

The function name along with the parameters list is known as the **function signature.** Care must be taken that while calling the method, the syntax of the function call must match with the function signature. Let's see an example for function call:

Example:

```
readint( );
```

Whenever the compiler comes across a function call, it takes the control of execution to the first statement in the function's definition. After the completion of function i.e., whenever the compiler comes across the **return** statement or the closing brace of the function's body, the control will return back to the next statement after the function call. Let's see this in the following example:



Calling function and Called function

The point at which the function is being invoked or called is known as the **calling function**. The function which is being executed due to the function call is known as the **called function.** Example for both calling function and the called function is given in the above example.

Formal Parameters and Actual Parameters

A parameter or argument is data which is taken as input or considered as additional information by the function for further processing. There are two types of parameters or arguments. The parameters which are passed in the function call are known as **actual parameters or actual arguments.** The parameters which are received by the called function are known as **formal parameters or formal arguments.** Example is shown below:

```
main( )
{
        int x = 10, y = 20;
        add(x, y);
}
void add(int a, int b)
{
        printf("Sum is: %d",(a+b));
}
```

In the above example, **x and y** are known as actual parameters and **a and b** are known as formal parameters. In the above code, we can see that the return type of the function **add** is **void.** This is a keyword in C. The **void** keyword is a datatype. If the function does not return any value, we specify the data type **void.** Generally **void** means nothing / no value.

Note: The formal parameters and actual parameters can have the same names i.e., if the actual parameters are x and y, then the formal parameters can also be x and y. But, it is recommended to use different names for actual and formal parameters.

Classification of Functions

Based on the parameters and return values, functions can be categorized into four types. They are:

1. Function without arguments and without return value.
2. Function without arguments and with return value.
3. Function with arguments and with return value.
4. Function with arguments and without return value.

Function without arguments and without return value

In this type of functions there are no parameters/arguments in the function definition and the function does not return any value back to the calling function. Generally, these types of functions are used to perform housekeeping tasks such as printing some characters etc.
Example:

```
void printstars( )

{

        int i;

        for(i = 0; i < 20; i++)

        {

                printf(" * ");

        }

        return;

}
```

In the above example, **printstars** function does not have any parameters. Its task is to print 20 stars whenever it is called in a program. Also **printstars** function does not return any value back.

Function without arguments and with return value

In this type of functions, the function definition does not contain arguments. But the function returns a value back to the point at which it was called. An example for this type of function is given below:

Example:

```
int readint( )

{

        int num;

        printf("Enter a number: ");

        scanf("%d",&num);

        return num;

}
```

In the above example, **readint** function has no parameters/arguments. The task of this function is to read a integer from the keyboard and return back to the point at which the function was called.

Function with arguments and with return value

In this type of functions, the function definition consists of parameters/arguments. Also, these functions returns a value back to the point at which the function was called. These types of functions are the most frequently used in programming. An example for this type of function can be seen below:

Example:

```
int add(int x, int y)

{

        int result;

        result = x + y;

        return result;

}
```

In the above example, the function **add** consists of two arguments or parameters **x** and **y**. The function adds both **x** and **y** and returns that value stored in the local variable **result** back to the point at which the function was called.

Predefined / Library Functions

A function is said to be a predefined function or library function, if they are already declared and defined by another developer. These predefined functions will be available in the library header files. So, if we want to use a predefined function, we have to include the respective header file in our program. For example, if we want to use **printf** function in our program, we have to include the **stdio.h** header file, as the function **printf** has been declared inside it.

Some of the header files in C are:

Header File	Description
<ctype.h>	Character testing and conversion functions
<math.h>	Mathematical functions
<stdio.h>	Standard I/O functions
<stdlib.h>	Utility functions
<string.h>	String handling functions
<time.h>	Time manipulation functions

Some of the predefined functions available in **ctype.h** header file are:

Function	Return Type	Use
isalnum(c)	int	Determine if the argument is alphanumeric or not
isalpha(c)	int	Determine if the argument is alphabetic or not
isascii(c)	int	Determine if the argument is ASCII character or not
isdigit(c)	int	Determine if the argument is a decimal digit or not.
toascii(c)	int	Convert value of argument to ASCII
tolower(c)	int	Convert character to lower case
toupper(c)	int	Convert letter to uppercase

Some of the predefined functions available in **math.h** header file are:

Function	Return Type	Use
ceil(d)	double	Returns a value rounded up to next higher integer
floor(d)	double	Returns a value rounded up to next lower integer
cos(d)	double	Returns the cosine of d
sin(d)	double	Returns the sine of d
tan(d)	double	Returns the tangent of d
exp(d)	double	Raise e to the power of d
fabs(d)	double	Returns the absolute value of d
pow(d1, d2)	double	Returns d1 raised to the power of d2
sqrt(d)	double	Returns the square root of d

Some of the predefined functions in **stdio.h** header file are:

Function	Return type	Use
getc(f)	int	Read a single character from file f
putc(c,f)	int	Write a single character to file
getchar(void)	int	Read a single character from standard input
putchar(void)	int	Write a single character to standard output
gets(s)	char*	Read string s from standard input
puts(s)	int	Write string s to standard output
printf(...)	int	Send data items to standard output

Some of the predefined functions in **stdlib.h** header file are:

Function	Return Type	Use
abs(i)	int	Return the absolute value of i
exit(u)	void	Close all file and buffers, and terminate the program
rand(void)	int	Return a random positive integer
calloc(u1, u2)	void*	Allocate memory for an array having u1 elements, each of length u2 bytes
malloc(u)	void*	Allocate u bytes of memory
realloc(p,u)	void*	Allocate u bytes of new memory to the pointer variable p
free(p)	void	Free a block of memory whose beginning is indicated by p

Some of the predefined functions in **string.h** header file are:

Function	Return Type	Use
strcmp(s1,s2)	int	Compare two strings
strcpy(s1,s2)	char*	Copy string s2 to s1
strlen(s)	int	Return the number of characters in string s
strrev(s)	char*	Return the reverse of the string s

Some of the predefined functions available in **time.h** header file are

Function	Return type	Use
difftime(11,12)	double	Return the difference between 11 ~ 12.
time(p)	long int	Return the number of seconds elapsed beyond a designated base time

Programs:

```
/* C program to demonstrate functions */
#include<stdio.h>
#include<conio.h>
int readint(); /*Function Declaration*/
main()
{
   int x;
   clrscr();
   x = readint();
   printf("Value of x is: %d",x);
}
int readint()  /* Function Definition */
{
   int n;
   printf("Enter a integer value: ");
   scanf("%d",&n);
   return n;
}
```

```
/* C program to add numbers by using a function */
#include<stdio.h>
#include<conio.h>
int addint();
main()
{
   int sum;
   clrscr();
   sum = addint();
   printf("Sum is: %d",sum);
   getch();
}
int addint()
{
   int a,b;
   printf("Enter the values of a and b: ");
   scanf("%d%d",&a,&b);
   return a+b;
}
```

```
/* C program to print 200 stars using a function */
#include<stdio.h>
#include<conio.h>
void printstars();
main()
{
   clrscr();
   printstars();
   getch();
}
void printstars()
{
   int i;
   for(i = 0; i < 200; i++)
   {
      printf("* ");
   }
}
```

```
/* C program to add two numbers by using a function with parameters */
#include<stdio.h>
#include<conio.h>
int addint(int x, int y);
main()
{
   int a, b, sum;
   clrscr();
   printf("Enter the values of a and b: ");
   scanf("%d%d",&a,&b);
   sum = addint(a, b);
   printf("Sum is: %d", sum);
   getch();
}
int addint(int x, int y)
{
   return x + y;
}
```

```
/* C program to print n number of stars using a function */
#include<stdio.h>
#include<conio.h>
void printstars(int n);
main()
{
   int n;
   clrscr();
   printf("Enter the value of n: ");
   scanf("%d",&n);
   printstars(n);
   getch();
}
void printstars(int n)
{
   int i;
   for(i = 0; i < n; i++)
   {
      printf("* ");
   }
}
```

```
/* C program to find whether the given number is even or odd using a function
*/
#include<stdio.h>
#include<conio.h>
void evenodd(int x);
main()
{
   int n;
   clrscr();
   printf("Enter a number: ");
   scanf("%d",&n);
   evenodd(n);
   getch();
}
void evenodd(int x)
{
   if(x % 2 == 0)
```

```
   {
      printf("Entered number is even");
   }
   else
   {
      printf("Entered number is odd");
   }
}

/* C program to find the factorial of a given number using a function */
#include<stdio.h>
#include<conio.h>
void factorial(int x);
main()
{
   int n;
   clrscr();
   printf("Enter the value of n: ");
   scanf("%d",&n);
   factorial(n);
}
void factorial(int x)
{
   int i, fact = 1;
   if(x == 0)
   {
      printf("Factorial of 0 is: 1");
   }
   else
   {
      for(i = 1; i <= x; i++)
      {
         fact = fact * i;
      }
      printf("Factorial of %d is : %d", x, fact);
   }
}
```

```
/* C program to generate the first n terms of the Fibonacci sequence using a function*/
#include<stdio.h>
#include<conio.h>
void fib(int n);
main()
{
   int number;
   clrscr();
   printf("Enter the number of terms: ");
   scanf("%d",&number);
   fib(number);
   getch();
}
void fib(int n)
{
   int i;
   int a = 0,b = 1,temp = 0;
   if(n == 1)
   {
      printf("%d",0);
   }
   else if(n == 2)
   {
      printf("%d %d ",0,1);
   }
   else
   {
      printf("%d %d ",a,b);
      for(i = 2; i < n; i++)
      {
         temp = a+b;
         a = b;
         b = temp;
         printf("%d ",b);
      }
   }
}
```

Nested Functions

A function calling another function within its function definition is known as a nested function. So, far we are declaring a **main** function and calling other user-defined functions and predefined functions like **printf, scanf, gets, puts** etc., So, **main** function can be treated as a nested function. Let's see the following example:

```
main()
{
   clrscr();
   func1();
   getch();
}
void func1()
{
   for(i = 1; i<= 10; i++)
   {
      func2();
   }
}
void func2()
{
   printf("%d\n",i);
}
```

In the above example, the **main** function is calling three functions namely: **clrscr**, **func1** and **getch**. So, **main** is a nested function. Also, in the definition of **func1**, it is calling another function **func2.** So, **func1** is also a nested function.

Note: *In programs containing nested functions, the enclosing or outer function returns back only when all the inner functions complete their task.*

Program:

```
/* C program to demonstrate a nested function */
#include<stdio.h>
#include<conio.h>
int i;
void func1();
void func2();
main()
{
   clrscr();
   func1();
   getch();
}
void func1()
```

```
{
   for(i = 1; i<= 10; i++)
   {
      func2();
   }
}
void func2()
{
   printf("%d\n",i);
}
```

Recursion

A function is said to be recursive, if a function calls itself within the function's definition. Let's see the following example:

```
void func1()
{
   int i = 0;
   i++;
   printf("%d\n",i);
   func1();   /*Recursive Call */
}
```

In the above example, func1 is calling itself in the last line of its definition. When we write recursive functions, the function only returns back to the main program when all the recursive calls return back.

Note: *When writing recursive functions, proper care must be taken that the recursive calls return a value back at some point. Otherwise, the function calls itself infinite number of times.*

Program:

```
/* C Program to demostrate recursion */
#include<stdio.h>
#include<conio.h>
void func1();
main()
{
```

```
   clrscr();
   func1();
   getch();
}
void func1()
{
   int i = 0;
   i++;
   printf("%d\n",i);
   func1();
}
```

```
/* C program to find the factorial of a given number using a recursive function
*/
#include<stdio.h>
#include<conio.h>
int factorial(int x);
main()
{
   int n, fact;
   clrscr();
   printf("Enter the value of n: ");
   scanf("%d",&n);
   fact = factorial(n);
   printf("Factorial of %d is: %d", n, fact);
}
int factorial(int x)
{
   if(x == 0)
   {
      return 1;
   }
   else
   {
      return x*factorial(x-1);
   }
}
```

```
/* C program to generate the first n terms of the Fibonacci sequence using a
recursive function*/
#include<stdio.h>
#include<conio.h>
int fib(int n);
main()
{
   int i, number;
   clrscr();
   printf("Enter the number of terms: ");
   scanf("%d",&number);
   if(number == 1)
   {
      printf("0");
   }
   else if(number == 2)
   {
      printf("0 1");
   }
   else
   {
      printf("0 1 ");
      for(i = 3; i <= number; i++)
      {
         printf("%d ", fib(i));
      }
   }
   getch();
}
int fib(int n)
{
   int i, temp;
   if(n == 1)
   {
      return 0;
   }
   else if(n == 2)
   {
      return 1;
   }
```

```
   else
   {
      temp = fib(n-1) + fib(n-2);
      return temp;
   }
}
```

Types of Variables

A variable is a memory location inside memory which is referred using a name. The value inside a variable changes throughout the execution of the program. Based on where the variable is declared in the program, variables can be divided into two types. They are:

1. Local Variables
2. Global Variables

Local Variables

A variable is said to be a local variable if it is declared inside a function or inside a block. The scope of the local variable is within the function or block in which it was declared. A local variable remains in memory until the execution of the function or block in which it was declared in completes. Let's see the following example:

```
main()

{

        int x;

        printf("x = %d",x);

}
```

In the above example, the variable **x** is a local variable with respect to the **main** function. Variable **x** is not accessible outside main function and **x** remains in the memory until the execution of the **main** function completes.

Global Variables

A variable is said to be a global variable if it is declared outside all the functions in the program. A global variable can be accessed throughout the program by any function. A global variable remains in the memory until the program terminates. In a multi-file program, a global can be accessed in other files wherever the variable is declared with the storage class **extern**.

Types of variables and their scope and lifetime can be summarized as shown below:

Type of Variable	Declaration Location	Scope (Visibility)	Lifetime (Alive)
Local Variable	Inside a function/block	Within the function/block	Until the function/block completes
Global Variable	Outside a function/block	Within the file and other files marked with extern	Until the program terminates

Program

```
/* C program to demonstrate local and global variables */
#include<stdio.h>
#include<conio.h>
int g = 10;
main()
{
   int x = 20;
   clrscr();
   printf("Inside main, g = %d",g);
   printf("\nInside main, x = %d",x);
   {
      int y = 30;
      printf("\nInside block, g = %d",g);
      printf("\nInside block, y = %d",y);
      printf("\nInside block, x = %d",x);
   }
   printf("\nOutside block, g = %d",g);
   /*printf("Outside block, y = %d",y);*/
   printf("\nOutside block, x = %d",x);
   getch();
}
```

In the above example, **g** is a global variable and **x** is a local variable with respect to main and **y** is a local variable within the block. The variable **y** cannot be accessed outside the block. That is why the **printf** statement outside the block accessing the value of the variable **y** has been commented out.

Storage Classes

The storage classes specify the scope and lifetime of a variable in a C program. The **scope (active)** specifies in which parts of the program is the variable accessible and the **lifetime (alive)** specifies how long a variable is available in the memory so that the program will be able to access that variable. There are four storage classes in C. They are:

1. auto
2. register
3. extern
4. static

The storage classes' **auto**, **register and static** can be applied to local variables and the storage classes' **extern** and **static** can be applied to global variables.

auto

When a variable is declared with the storage class **auto,** the variable's scope is within the function or block in which it is declared and the lifetime is until the function or block in which it is declared completes. Syntax for declaring **auto** variable is shown below:

```
auto datatype variablename;
```

In any program, if a local variable is declared without any storage class then it is automatically set to **auto** storage class.

register

When a variable is declared with the storage class **register**, the variable will be stored inside one of the registers of the CPU. The registers are under the direct control of CPU. So, data inside the register can be processed at a faster rate than the data that resides in the main memory. For a program to execute faster, it is always best to store the most frequently used data inside register. The scope and lifetime of a **register** variable is same as that of a **auto** variable. Syntax for declaring a **register** variable is as shown below:

```
register datatype variablename;
```

extern

The **extern** storage class specifies that the variable is declared in some part of the program. Generally this storage class is used to refer global variables in a program. Note that **extern** variables cannot be initialized. The scope of a **extern** variable is throughout the entire program and the lifetime is until the program completes its execution.

In a multi-file program, a global variable in one file can be accessed from another file by using the storage class **extern.** Syntax for declaring a **extern** variable is as shown below:

```
extern datatype variablename;
```

static

The **static** storage class can be applied to both local variables and global variables. The **static** local variables are accessible only within the function or block in which they are declared, but their lifetime is throughout the program. The **static** global variables are accessible throughout the file in which they are declared but not in other files. Syntax for declaring **static** variable is shown below:

```
static datatype variablename;
```

The four storage classes can be summarized as shown below:

Storage Class	Declaration Location	Scope (Visibility)	Lifetime (Alive)
auto	Inside a function/block	Within the function/block	Until the function/block completes
register	Inside a function/block	Within the function/block	Until the function/block completes
extern	Outside all functions	Entire file plus other files where the variable is declared as extern	Until the program terminates
static (local)	Inside a function/block	Within the function/block	Until the program terminates
static (global)	Outside all functions	Entire file in which it is declared	Until the program terminates

Note: *The* ***extern*** *variables cannot be initialized. The default value for* ***static*** *variables is zero.*

Programs:

```
/* C program to demostrate auto storage class */
#include<stdio.h>
#include<conio.h>
void func1();
main()
{
   /* x is local variable with respect to main function */
   auto int x;
   clrscr();
   x = 20;
   printf("\nValue of x is: %d",x);
   func1();
   getch();
}
void func1()
{
   /*Since x is a auto or local variable of function main, it is not
     accessible in func1*/
   x = 10;
   printf("\nValue of x is: %d",x);
}
```

```
/* C program to demostrate register storage class */
#include<stdio.h>
#include<conio.h>
void printx();
main()
{
   clrscr();
   printx();
   getch();
}
void printx()
{
   register int i;
   for(i = 1;i <= 10000;i++)
   {
      printf("%d ",i);
   }
}
```

```
/* C program to demostrate global variables */
#include<stdio.h>
#include<conio.h>
int x;
void func1();
void func2();
void func3();
main()
{
   clrscr();
   x = 10;
   printf("x = %d\n",x);
   func1();
   printf("x = %d\n",x);
   func2();
   printf("x = %d\n",x);
   func3();
   printf("x = %d\n",x);
   getch();
```

```
}
void func1()
{
   x++;
}
void func2()
{
   x++;
}
void func3()
{
   int x = 10;
}
```

```
/* C program to demostrate extern storage class */
#include<stdio.h>
#include<conio.h>
void func1();
void func2();
void func3();
main()
{
   extern int x;
   clrscr();
   x = 10;
   printf("x = %d\n",x);
   func1();
   printf("x = %d\n",x);
   func2();
   printf("x = %d\n",x);
   func3();
   printf("x = %d\n",x);
   getch();
}
void func1()
{
   extern int x;
   x++;
```

```
}
int x;
void func2()
{
   x++;
}
void func3()
{
   int x = 10;
   x++;
}
```

```
/* C program to demostrate extern storage class in multiple files */
#include<stdio.h>
#include<conio.h>
#include"extf2.c"
int x;
main()
{
   x = 10;
   func1();
   func2();
   getch();
}
```

extf1.c

```
/* C program to demostrate extern storage class in multiple files */
void func1()
{
   extern int x;
   printf("\nx = %d",x);
}
void func2()
{
   int x = 20;
   printf("\nx = %d",x);
}
```

extf2.c

```
/* C program to demostrate static storage class - local variables */
#include<stdio.h>
#include<conio.h>
int count();
main()
{
   clrscr();
   printf("Value is: %d\n",count());
   printf("Value is: %d\n",count());
   printf("Value is: %d\n",count());
   getch();
}
int count()
{
   static int x = 0;
   x++;
   return x;
}
```

statg1.c

```
/* C program to demostrate static storage class - global variables */
#include<stdio.h>
#include<conio.h>
#include"statg2.c"
static int x;
void func1();
main()
{
   x = 10;
   printf("x = %d\n",x);
   func1();
   func2();
}
void func1()
{
   x++;
   printf("x = %d\n",x);
}
```

statg2.c

```
/* C program to demonstrate static storage class - global variables */

void func2()
{
   x++;
   printf("x = %d\n",x);
}
```

Passing arrays to functions

Passing one-dimensional arrays

We can also pass arrays as parameters to the called function. While passing one-dimensional array to a function, you should follow three rules. They are:

1. In the function declaration you should write a pair of square brackets [ ] beside the name of the array. No need to specify the size of the array.
2. In the function definition you should write a pair of square brackets [ ] beside the name of the array. Again no need to specify the size of the array.
3. In the function call, it is enough to just pass the array name as the actual parameter. No need to write the square brackets after the array name.

When an array is passed as an actual parameter, the formal parameter also refers to the same array which is passed as an actual parameter. When passing an array as a parameter, you are passing the address of the array, not the values in the array. So, if you make changes in the array using the formal name of the array, the changes are also reflected on the actual array.

Programs:

```
/* C program to find the largest number in a group of numbers using a function
*/
#include<stdio.h>
#include<conio.h>
void largest(int a[], int n);
main()
{
   int a[5], i;
   clrscr();
   for(i = 0; i < 5; i++)
   {
      printf("Enter a[%d]: ",i+1);
      scanf("%d",&a[i]);
   }
   largest(a,5);
   getch();
}
void largest(int a[], int n)
{
   int i, max;
   max = a[0];
   for(i = 1; i < n; i++)
   {
      if(max<a[i])
      {
```

```
            max = a[i];
        }
    }
    printf("The largest number is: %d",max);
}
```

/* C program to find the smallest number in a group of numbers using a function */

```
#include<stdio.h>
#include<conio.h>
void smallest(int a[], int n);
main()
{
    int a[5], i;
    clrscr();
    for(i = 0; i < 5; i++)
    {
        printf("Enter a[%d]: ",i+1);
        scanf("%d",&a[i]);
    }
    smallest(a,5);
    getch();
}
void smallest(int a[], int n)
{
    int i, min;
    min = a[0];
    for(i = 1; i < n; i++)
    {
        if(min>a[i])
        {
            min = a[i];
        }
    }
    printf("The smallest number is: %d",min);
}
```

Passing two-dimensional arrays

We can also pass two-dimensional arrays as parameters to a function. While passing two-dimensional arrays as parameters you should keep in mind the following things:

1. In the function declaration you should write two sets of square brackets after the array name. You should specify the size of the second dimension i.e., the number of columns.
2. In the function call you should write two sets of square brackets after the array name. Also you should specify the size of the second dimension i.e., the number of columns.
3. In the function call, it is enough to pass the name of the array as a parameter. No need to mention the square brackets.

Program:

```
/* C program to pass a two dimensional array as a parameter to a function */
#include<stdio.h>
#include<conio.h>
void printmatrix(int a[][3],int m,int n);
main()
{
   int a[3][3],i,j;
   clrscr();
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 3; j++)
      {
         printf("Enter a[%d][%d]: ",i,j);
         scanf("%d",&a[i][j]);
      }
   }
   printmatrix(a,3,3);
   getch();
}
void printmatrix(int a[][3],int m,int n)
{
   int i, j;
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 3; j++)
      {
         printf("%d ",a[i][j]);
      }
      printf("\n");
   }
}
```

```
/*C program to insert a sub-string into a string at a specified position*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
void strinst(char x[],char y[], int loc);
main()
{
   char str[20], substr[20];
   int pos;
   clrscr();
   printf("Enter the string: ");
   gets(str);
   printf("Enter the substring: ");
   gets(substr);
   printf("Enter the position number: ");
   scanf("%d",&pos);
   strinst(str, substr, pos);
   getch();
}
void strinst(char x[],char y[], int loc)
{
   char result[40];
   int i, j, k;
   for(i=0; i<loc; i++)
   {
      result[i] = x[i];
   }
   for(j=0, k=i; j<strlen(y); j++, k++)
   {
      result[k] = y[j];
   }
   for(k=i+strlen(y); k<strlen(x)+strlen(y); k++,i++)
   {
      result[k] = x[i];
   }
   result[k] = '\0';
   puts(result);
}
```

```
/* C program to delete n characters from the specified position in a string */
#include<stdio.h>
#include<conio.h>
#include<string.h>
void strdel(char x[],int num,int loc);
main()
{
   char str[20];
   int n, pos;
   clrscr();
   printf("Enter a string: ");
   gets(str);
   printf("How many characters you want to delete? ");
   scanf("%d",&n);
   printf("Enter the position: ");
   scanf("%d",&pos);
   strdel(str,n,pos);
   getch();
}
void strdel(char x[],int num,int loc)
{
   char result[20];
   int i,j;
   for(i=0; i<loc; i++)
   {
      result[i] = x[i];
   }
   for(j=i,i=i+num; j<(strlen(x)-num); j++,i++)
   {
      result[j] = x[i];
   }
   result[j] = '\0';
   puts(result);
}
```

```
/*C program to replace a character at beginning or ending or at the specified location in a string */
#include<stdio.h>
#include<conio.h>
#include<string.h>
void strrepb(char x[], char c);
void strrepe(char x[], char c);
void strrep(char x[], char c, char loc);
main()
{
   char str[10], ch;
   int choice, pos;
   /*clrscr();*/
   printf("Enter a string: ");
   gets(str);
   while(1)
   {
      printf("1. Replace a character at the begining of the string\n");
      printf("2. Replace a character at the ending of the string\n");
      printf("3. Replace a cahracter at specific position\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
            fflush(stdin);
            printf("Enter the character: ");
            scanf("%c",&ch);
            strrepb(str, ch);
            break;
         case 2:
            fflush(stdin);
            printf("Enter the character: ");
            scanf("%c",&ch);
            strrepe(str, ch);
            break;
         case 3:
            printf("Enter the position: ");
            scanf("%d",&pos);
```

```
            fflush(stdin);
            printf("Enter the character: ");
            scanf("%c",&ch);
            strrep(str, ch, pos);
            break;
         case 4:
            exit(0);
         default:
            printf("Invalid option. Try again...\n");
            break;
      }
   }
}
void strrepb(char x[], char c)
{
   x[0] = c;
   puts(x);
   printf("\n");
}
void strrepe(char x[], char c)
{
   x[strlen(x)-1] = c;
   puts(x);
}
void strrep(char x[], char c, char loc)
{
   x[loc-1] = c;
   puts(x);
}
```

Preprocessor Directives

C provides many features like structures, unions and pointers. Another unique feature of the C language is the preprocessor. The C preprocessor provides several tools that are not present in other high-level languages. The programmer can use these tools to make his program easy to read, easy to modify, portable and more efficient.

The preprocessor is a program that processes the source code before it passes through the compiler. Preprocessor directives are placed in the source program before the main line. Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions are taken and then the source program is handed over to the compiler.

All the preprocessor directives follow special syntax rules that are different from the normal C syntax. Every preprocessor directive begins with the symbol **#** and is followed by the respective preprocessor directive. The preprocessor directives are divided into three categories. They are:

1. Macro Substitution Directives
2. File Inclusion Directives
3. Compiler Control Directives

Macro Substitution Directives

Macro substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. The preprocessor accomplishes this task under the direction of **#define** statement. This statement, usually known as a *macro definition* takes the following form:

```
#define  identifier  string
```

If this statement is included in the program at the beginning, then the preprocessor replaces every occurrence of the **identifier** in the source code by the string.

Note: *Care should be taken that there is no space between the **#** and the word **define**. Also there should be atleast a single space between **#define** and the **identifier** and between the **identifier** and the **string**. Also, there will be no semi-colon at the end of the statement.*

There are different forms of macro substitution. The most common are:

1. Simple macro substitution
2. Argumented macro substitution
3. Nested macro substitution

Simple Macro Substitution

The simple macro substitutions are generally used for declaring constants in a C program. Some valid examples for simple macro substitution are:

```
#define     PI      3.1412
#define     MAX     100
#define     START   main() {
#define     STOP    }
```

Whenever the preprocessor comes across the simple macros, the identifier will be replaced with the corresponding string. For example, in a C program, all the occurrences of PI will be replaced with 3.1412.

Argumented Macro Substitution

The preprocessor allows us to define more complex and more useful form of substitutions.

The Argumented macro substitution takes the following form:

```
#define   identifier(arg1, arg2, ..... , argn)   string
```

Care should be taken that there is no space between the identifier and the left parentheses. The identifiers arg1, arg2, …. , argn are the formal macro arguments that are analogous to the formal arguments in a function definition. In the program, the occurrence of a macro with arguments is known as a **macro call**. When a macro is called, the preprocessor substitutes the string, replacing the formal parameters with actual parameters.
For example, if the Argumented macro is declared as shown below:

```
#define CUBE(x) (x*x*x)
```

and the macro is called as shown below:

```
volume = CUBE(side);
```

Then the preprocessor will expand the above statement as:

```
volume = (side * side * side);
```

Nested Macro Substitution

We can use one macro inside the definition of another macro. Such macros are known as nested macros. Example for a nested macro is shown below:

```
#define SQUARE(x) x*x
#define CUBE(x) SQUARE(x) * x
```

Programs

```
/* C program to demonstrate simple macro substitution */
#include<stdio.h>
#include<conio.h>
#define MAX 100
main()
{
   int n;
   clrscr();
   printf("Enter the value of n: ");
   scanf("%d",&n);
   if(n < MAX)
   {
      printf("%d is less than MAX",n);
   }
   else
   {
      printf("%d is greater than MAX",n);
   }
   getch();
}
/* C program to demonstrate argumented macro substitution */
#include<stdio.h>
```

```
#include<conio.h>
#define SQUARE(x) x*x
main()
{
   int i;
   clrscr();
   for(i = 1; i <= 10; i++)
   {
      printf("%d\n",SQUARE(i));
   }
   getch();
}
```

```
/* C program to demonstrate nested macro substitution */
#include<stdio.h>
#include<conio.h>
#define SQUARE(x) x*x
#define AREA(x) 3.14*SQUARE(x)
main()
{
   int r;
   clrscr();
   printf("Enter the radius: ");
   scanf("%d",&r);
   printf("Area of the circle is: %f",AREA(r));
   getch();
}
```

File Inclusion Directives

The external files containing functions or macro definitions can be linked with our program so that there is no need to write the functions and macro definitions again. This can be achieved by using the **#include** directive. The syntax for this directive is as shown below:

```
#include “filename”
OR
#include<filename>
```

We can use either of the above statements to link our program with other files. If the **filename** is included in double quotes, the file is searched in the local directory. If the **filename** is included in angular brackets, then the file is searched in the standard directories.

Compiler Control Directives

Following are the compiler control directives:

Directive	Purpose
#ifdef	Test for a macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether a macro is not defined
#if	Test a compile-time condition
#else	Specifies alternative when #if fails

These compiler control directives are used in different situations. They are:

Situation 1

You have included a file containing some macro definitions. It is not known whether a certain macro has been defined in that header file. However, you want to be certain that the macro is defined.

This situation refers to the conditional definition of a macro. We want to ensure that the macro TEST is always defined, irrespective of whether it has been defined in the header file or not. This can be achieved as follows:

```
#include”DEFINE.H”
#ifndef TEST
#define TEST 1
#endif
```

DEFINE.H is the header that is supposed to contain the definition of **TEST** macro. The directive **#ifndef TEST** searches the definition of **TEST** in the header file and if it is not defined, then all the lines between the **#ifndef** and the corresponding **#endif** directive are executed in the program.

Situation 2

Suppose a customer has two different types of computers and you are required to write a program that will run on both the systems. You want to use the same program, although a certain lines of code must be different for each system.

The main concern here is to make the program portable. This can be achieved as shown below:

```
#ifdef IBM_PC
{
        ----
        ----
}
#else
{
        ----
        ----
}
#endif
```

If we want to run the program on a IBM PC, we include the directive **#define IBM_PC,** otherwise we won't.

Situation 3

You are developing a program for selling in the open market. Some customers may insist on having certain additional features. However, you would like to have a single program that would satisfy both types of customers.

This situation is similar to the above situation and therefore the control directives take the following form:

```
#idef ABC
        Group-A lines
#else
        Group-B lines
#endif
```

Group-A lines are included if the customer ABC is defined. Otherwise, Group-B lines are included.

Situation 4

Suppose if you want to test a large program, you would like to include print calls in the program in certain places to display intermediate results and messages in order to trace the flow of execution and errors.

For this purpose we can use **#if** and **#else** directive as shown below:

```
#if constant expression
{
        Statement 1;
        Statement 2;
        -----
}
else
{
        Statement 1;
        Statement 2;
        -----
}
#endif
```

Programs

```
/* C program to demostrate #undef preprocessor directive */
#include<stdio.h>
#include<conio.h>
#define NAME "teja"
main()
{
   clrscr();
   puts(NAME);
   #undef NAME
   puts(NAME);
   getch();
}
```

```
/* C program to demostrate #ifdef and #endif preprocessor directives */
#include<stdio.h>
#include<conio.h>
#define MAX 100
main()
{
   clrscr();
   #ifdef MAX
   #define COUNT 10
   #endif
   printf("COUNT = %d",COUNT);
   getch();
}
```

```
/* C program to demostrate #ifndef and #endif preprocessor directives */
#include<stdio.h>
#include<conio.h>
#ifndef MAX
#define MAX 100
#endif
main()
{
   clrscr();
   printf("MAX = %d",MAX);
   getch();
}
```

```
/* C program to demostrate #if and #else preprocessor directives */
#include<stdio.h>
#include<conio.h>
#define MAX 100
main()
{
   clrscr();
   #ifdef MAX
   {
      printf("MAX is defined");
   }
   #else
   {
      printf("MAX is not defined");
   }
   #endif
   getch();
}
```

ANSI Preprocessor Directives

The ANSI committee has added some more preprocessor directives to the existing list. They are:

Directive	Purpose
#elif	Provides alternative test facility
#pragma	Specifies compiler instructions
#error	Stops compilation when an error occurs

#elif Directive

The **#elif** directive enables us to establish an "if…else…if" sequence for testing multiple conditions. The syntax is as shown below:

```
#if expr1
        Stmts;
#elif  expr2
        Stmts;
#elif  expr3
        Stmts;
#endif
```

#pragma Directive

The **#pragma** directive is an implementation oriented directive that allows the user to specify various instructions to be given to the compiler. Syntax is as follows:

```
#pragma name
```

Where **name** is the name of the pragma we want. For example, under Microsoft C, **#pragma loop_opt(on)** causes loop optimization to be performed.

#error Directive

The **#error** directive is used to produce diagnostic messages during debugging. The general format is:

```
#error  error-message
```

When the **#error** directive is encountered by the compiler, it displays the error message and terminates the program.
Example:

```
#if !defined(FILE_G)
#error NO GRAPHICS FILE
#endif
```

Preprocessor Operations

Stringizing Operator

ANSI C provides an operator **#** called stringizing operator to be used in the definition of macro functions. This operator converts a formal argument into a string. For example, if the macro is defined as follows:

```
#define    sum(xy)    printf(#xy  " = %f \n", xy)
```

and somewhere in the program the statement is written as: **sum(a+b);** then the preprocessor converts this line as shown below:

```
printf("a+b"  " = %f \n", a+b);
```

Token Pasting Operator

The token pasting operator **##** defined by ANSI enables us to combine two tokens within a macro definition to form a single token.

Programs

```
/*C program to demonstrate Stringizing operator # */
#include<stdio.h>
#include<conio.h>
#define sum(x) printf(#x"= %d",x);
main()
{
   int a,b;
   clrscr();
   a = 10, b = 20;
   sum(a+b);
   getch();
}
```

```
/* C program to demonstrate preprocessor directives */
#include<stdio.h>
#include<conio.h>
#define START main() {
#define STOP }
#define PRINT(x) printf(#x)
#define COMBINE(x,y,z) x##y##z
#define hai "hai"
START
   clrscr();
   printf(COMBINE(h,a,i));
   PRINT(\nhello world);
   getch();
STOP
```