CHAPTER 4

FUNCTIONS

4.1 Introduction

Functions are the building blocks of C++ programs. Functions are also the executable segments in a program. The starting point for the execution of a program is main ( ). Functions are advantageous as they

- ✓ reduce the size of the program
- ✓ induce reusability of code

```
// To print X!/F! – Program - 4.1
# include < iostream.h >
# include <conio.h >
void main ( )
{  int x, f,xfact=1, ffact = 1;
clrscr ( );
cout << "\nEnter values ...";
cin >> x >> f;   for (int a =1; a < = x;
a ++)
xfact * = a;
for (a = 1; a< = f; a++)
ffact * = a;
cout << xfact/ffact;
getch();
}
```

```
// To print X!/F! using functions
// Program - 4.2
# include <iostream.h >
# include <conio.h >
int fact (int num)
{
int factorial = 1;
for (int a=1; a<= num; a++)
 factorial *= a;
return factorial;
}
void main ( )
{   int x, f;
   clrscr ( );
   cout<<"\nEnter values…";
    cin >> x >> f;
   cout << fact (x)/fact(f);
  }
```

In Program –4.1, the code for Factorial evaluation is repeated twice. In the Program –4. 2, the function **fact (int num)** is invoked whenever required. Functions thus encourage :

- ✓ Reusability of code (function fact is executed more than once)
- ✓ A function can be shared by other programs by compiling it separately and loading them together.

The general syntax showing the various blocks of a function :

```
// To print X!/F! using functions
// Program - 4.3
# include <iostream.h >
# include <conio.h >
int fact (int num)
{
  int factorial = 1;
  for (int a=1; a<= num; a++)
        factorial *= a;
  return factorial;
}

void main ( )
{   int x, f;
    clrscr ( );
    cout<<"\nEnter values…";
    cin >> x >> f;
    cout << fact (x)/fact(f);
}
```

Prototype with arguments

return statement marks the end of the function and also transfers control to the statement after call statement

Statements to invoke function fact – call statement

4.2 Function Prototyping

Functions should be declared before they are used in a program. Declaration of a function is made through a function prototype.

For example look at the Program –4. 4.

```
//Program -4.4
# include <iostream.h>
   void fun (char name [ ]) ;
  void main ( )
  {
    char n [ ] = { "C++ programming…."};
    fun (n);
  }
 void fun (char name[] )
{ cout << name; }
```

function prototype (declaration of function)

Function Definiton

The prototype provides the following information to the compiler

1. Number and type of arguments -(char name [ ] - is an argument)

2. The type of return values (in the above example fun does not have any return value, as the data type of the function is void. Recall Program –2 , in which the return type is int for the function Fact ( ))

The general syntax of a function prototype

<type > <function identifier > <arguments);

For example :

```
void fun (char);
int max (int, int);
int max (int a, int b);
```

The main purpose of function prototype is to help the compiler to check the data requirement of the function. With function prototyping, a template is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching of the arguments or the return types will be treated as errors by compiler, and flagged at the time of compilation.

Why do you think the prototype int max (int, int) is valid??

In a function declaration, the names of the arguments are dummy variables and therefore they are optional. The variables in the prototype act as place holders.

The arguments' names are required in function definition, as the arguments are referenced inside the function.

```
void fun (char name [ ] )
{ cout << name; }
int fact (int num)
{ int f = 1;
  for (int a = 1;, a < = num; a ++)
fx = a;
return f;
}
```

The argument num is referenced inside function.

4.3 Calling a Function

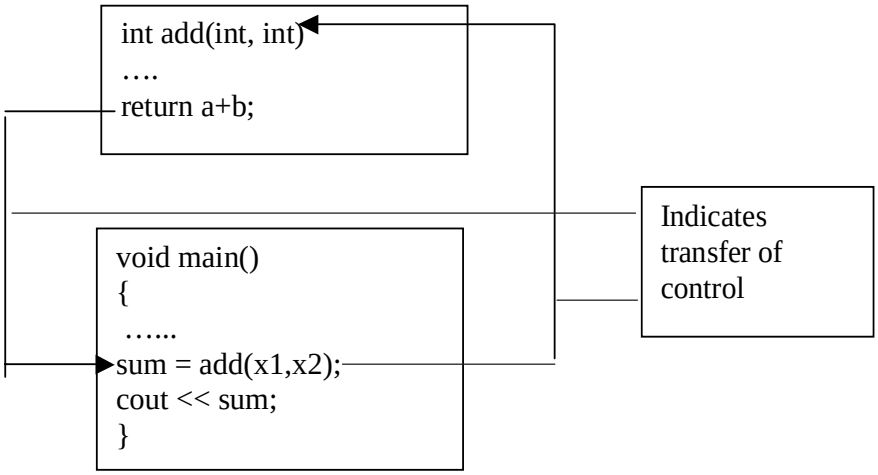
A function can be called or invoked from another function by using its name. The function name may include a set of actual parameters, enclosed in parentheses separated by commas. For example,

```
// Program - 4.5
# include <conio.h>
# include <iostream.h>

int add (int a, int b)

{ return a + b;}
void main ( )
{ int x1, x2, sum = 0
  cin >> x1 >> x2;

   sum = add (x, x2);
  cout << sum;
}
```

formal parameters

actual parameter

Working of a function :



4.4 Parameter Passing in Functions

The call statement communicates with the function through arguments or parameters.

> Parameters are the channels through which data flows from the call statement to the function and vice versa.

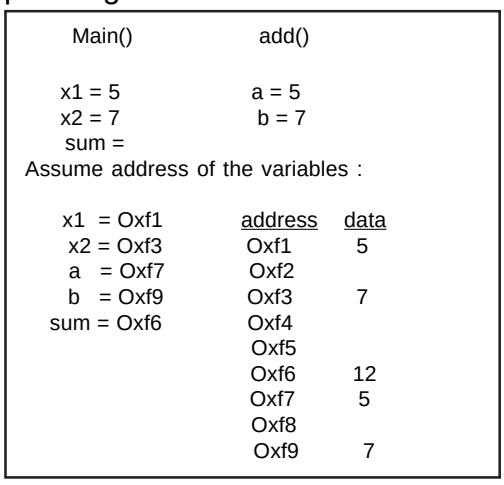
In C++, functions that have arguments can be invoked by

- ✓ Call by value
- ✓ Call by reference

4.4.1 Call by Value

In this method, the called function creates new variables to store the value of the arguments passed to it. This method copies the values of actual parameters (parameters associated with call statement) into the formal parameters (the parameters associated with function header), thus the function creates its own copy of arguments and then uses them. Recall the example Program - 4.5

```
// Program - 4.5
# include <iostream.h>
# include <conio.h>
int add (int a, int b)
{ return a + b;}
void main ( )
{ int x1, x2, sum;
cin >> x1 >> x2;
sum = add (x, x2);
cout << sum;
}
Assume x1 = 5, x2 = 7
```



Have you noticed that the actual parameters x1 and x2 and the formal parameters a&b have been allocated different memory locations? Hence, in call by value method, the flow of data is always from the call statement to the function definition.

```
// Program - 4.6
// To exchange values
#include <iostream.h>
#include <conio.h>
# include <iomanip.h>
void swap (int n1, int n2)
{   int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
    cout << '\n'<<n1<<'\t'<<n2<<'\n';
}
```

```
void main ( )
{
   int m1 = 10, m2 = 20;
   clrscr ( );
   cout <<"\n Values before invoking swap"
        << m1 << '\t' << m2;
  cout << "\n Calling swap..";
  swap (m1, m2);
  cout << "\n Back to main.. Values are"
       << m1 << '\t' << m2;
  getch ( );
}
```

Output

Values before invoking swap 10 20
Calling swap …..
20 10
Back to main…… Values are 10 20

Why do you think the exchange of values of the variables m1 and m2 are not reflected in the main program??

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 these arguments are copied into the newly created variables. Hence, changes or modifications that are made to formal parameters are not reflected in the actual parameters.

> In call by value method, any change in the formal parameter is not reflected back to the actual parameter.

4.4.2 Call by reference

In this method, the called function arguments - formal parameters become alias to the actual parameters in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Recall the example Program 4.6. Let us now rewrite the function using reference parameters.

```
//Program - 4.7
// To exchange values

# include <iostream.h>
#include <conio.h>
void swap (int &n1, int &n2)
{
  int temp;
  temp = n1;
  n1 = n2;
  n2 = temp;
  cout<<'\n'<< n1
        <<'\t'<<n2<<'\n';
}
```

```
void main ( )
{
   int m1 = 10, m2 = 20;
   clrscr();
   cout<<"\nValues before swap call"
         << '\t' << m1 << '\t' << m2;
   swap(m1,m2);
   cout<<"\n Calling swap..";
   cout<<"\n Back to main.Values are"
         << '\t' << m1 << '\t'<< m2;
   getch ( );
}
```

Output:

Values before invoking swap 10 20
Calling swap..
20 10
Back to main. Values are 20 10

The modifications made to formal parameters are reflected in actual parameters, because formal and actual parameters in reference type point to the same storage area.

Look at the following depiction:

Main	**Swap**
m1 = 10	n1 = 10
m2 = 20	n2 = 20
	temp

Assume storage area of m1 is Oxf1, and m2 is Oxf4.

m1 = Oxf1 = 10

m2 = Oxf4 = 20

Reference to formal parameters may be read as

n1 = 10; n1 is a reference to m1, which may be depicted as:

int &n1 = m1

This means that n1 is an alias to m1, hence m1 and n1 refer to same storage area, hence the statements may be rewritten as :

n1 = m1 = Oxf1 = 10
n2 = m2 = Oxf4 = 20

Address	Before Exchange	After exchange
Oxf1 (n1, m1)	10	20
Oxf4 (n2, m2)	20	10

Hence, changes made to formal parameters are reflected in actual parameters.

> In call by reference method, any change made in the formal parameter is reflected back in the actual parameter.

Try out

```
// Reference variables
// Program -4.8
        # include <iostream.h>
        # include <conio.h>
        void main ( )
        {
           int num1 = 10, & num2 = num1;
            num2 ++;
              cout << num1;
        }
```

Output displayed will be **11**

By virtue of reference num1 and num2 point to the same storage location.
Hence, change of value in num1 is reflected in num2.

Rules for actual parameters:

1. The actual parameters can be passed in the form of constants or variables or expressions to the formal parameters which are of value type.

For example,

For a function prototype : int add (int n1, int n2); - the call statements may be as follows :

x = add (5, 10);
x = add (a1, a2); where a1 and a2 are variables

2. The actual parameters can be passed only as variables to formal parameters of reference type.

For example,

int add (int & n1, int & n2);
x = add (a1, b1) ; where a1 and b1 are variables

The following call statements are invalid:

x = add ((a1 + b1), a1);
x = add (5,10);

```
// Program - 4.9
//To print 5 stars per row and 5 such rows
# include <iostream.h>
# include <conio.h>

void fun_starts (int &i)
{
 int j = 5;
 for (i= 1; i <= j; i++)
      cout << ' '<<'*';
}

void main ( )
{
  int mi = 1;
  clrscr( );
  for (; mi<= 5; mi++)
  {
      cout << '\n';
      fun_starts (mi);
  }
   getch ( );
      }

}
```

Why does the above program does not behave the way to produce the result as mentioned in comment line?

The output produced is :

```
* * * * *
```

Reason – the variable i is a reference to the variable mi.Since the variable i gets a value 6 in the function, mi is also automatically updated to 6, hence, the 'for' loop in main ( ) is executed only once.

4.3.4 Default arguments

In C++, one can assign default values to the formal parameters of a function prototype.

For example :

```
// Program - 4.10
// formal parameters with default values
# include <iostream.h>
# include <conio.h>
float power (float n, int p = 1)
{
  float prd = 1;
  for (int i = 1;  i<= p; i++)
      prd *= n;
  return prd;
}

void main ( )
{
   clrscr ( );
   int x = 4, b = 2;
   cout << "\n Call statement is power(b, x)..."
         << power (b, x);
   cout << "\n Call statement is power(b).. "
         << power (b);
   getch ( );
}
```

Output:

Call statement is power (b, x) .. 16
Call statement is power (b) .. 2

In the call statement power (b,x), initialization is

n= b, p = x

In the second form power (b), the variable n is initialized, whereas p takes the value 1 (default argument), as no actual parameters is passed.

NOTE:

- ✓ The default value is given in the form of variable initialization.
- ✓ The default arguments facilitate the function call statement with partial or no arguments.
- ✓ The default values can be included in the function prototype form right to left, i.e., we cannot have a default value for an argument in between the argument list.

Try out the following Program.

```
//Program - 4.11
# include <iostream h>
# include <conio.h>
int area (int side1 = 10, int side2=20
{ return (side1 * side 2); }

void main ( )
{   int s1 = 4, s2 = 6;
    clrscr ( ) ;
    cout << area (s1, s2) << '\n';
    cout << area (s1) << '\n';
    cout << area (s2) << '\n';
    getch ( );
}
```

```
Output:
 24
 80
 120
Variable initialization
I form - side1 = s1,
side2 = s2
II form - side1 = s1
III form - side1 = s2
```

What will be the output of the following program?

```
// Program - 4.12
// arguments with default values

# include <iostream.h>
# include <conio.h>

void print (int times, char ch = ' * ')
{
  cout << '\n';
  for (int i = 1, i < = times; i ++)
     cout << ch;
}
void main ( )
{
   clrscr ( );
    print (50);
    print ('A', 97);
    print ( );
}
```

Solution:

print (50) - 50 is assigned to the argument times. Hence, the output ' * ' will be printed 50 times.

print ('A', 97) - 'A' is assigned to argument times (implicit conversion of character to integer takes place). Hence, times gets the value as 65.

The constant 97 is assigned to ch, hence ch gets the value as ‘a’.

The actual parameters are matched with formal parameters on the basis of one- to -one correspondence.
Hence, 65 times, ‘a’ will be printed.

print ( ) - In the absence of actual arguments, the formal parameters takes the default arguments. Hence, the output will be displayed as ‘ * ‘ - 50 times.

4.5 Returning Values

The functions that return no value is declared as void. The data type of a function is treated as int, if no data type is explicitly mentioned. For example,

```
int add (int, int);
add (int, int);
```

In both prototypes, the return value is int, because by default the return value of a function in C++ is of type int.

Look at the following examples:

Sl.No.	Function Prototype	Return type
1	float power (float, int)	float
2	char choice ( )	char
3	char * success ( )	pointer to character
4	double fact (int)	double

4.5.1 Returning by reference

Reference or alias variables:

```
// Program -  4.13
# include <iostream.h>
# include <conio.h>
void main ( )
{    int i = 5;
     int  &count = i ;
     cout << "\nCount: " << count;
     count ++;
     cout << "\ni: " << i;
     getch ( );
}
```

Output:

```
Count: 5
 i: 6
```

Why do you think count gets the value as 5?

Why is the variable i updated to 6, when count was incremented???

The reason being that the variables count and i refer to the same data in the memory. Reference variables also behave the same way.

Using this principle, try and find out as to what will be output of the following program:

```
// Program - 4.14
# include <iostream h>
# include <conio.h>
int &maxref (int &a, int &b)
{ if (a>b)
      return a;
   else
       return b;
}
void main ( )
{ int x = 20, y = 30, max = 0;
     max = maxref (x,y);
      cout << "\n Maximum is: " << max;
}
```

Output

Maximum is : 30

In the above program, the function maxref returns a reference to int type of variable. The function call maxref (x,y) will return a reference to either a or b depending upon which one is bigger of the two. Hence, the variable max gets the value of the variable y.

What will be the output of the following program?

```
// Program 4.15
# include <iostream h>
# include <conio.h>
int & maxref (int & a, int & b)
{ if (a > b),
return a;
   else
     return b;
}
void main ( )
{ int x = 20, y = 30, max = 0;
     maxref (x,y) = -1;
      cout << "\n Value of x is : " << x;
      cout << "\n Value of y is: " <<y;
       getch ( );
}
```

Output

```
Value of x is  :    20
Value of y is  :    -1
```

NOTE:

1. A function returning a reference can appear on the left-hand side of an assignment.
2. In the above example, the variable y gets the value -1, since the function maxref. establishes reference with the formal parameter b, whose corresponding variable in main block is ‘y’.
3. The formal parameters for a reference function should always be of reference parameter type in the sense -

   int & maxref (int a, int b);
   will yield compilation error, as the scope of the variables a&b
   are within the function block maxref.

4.6 Inline Functions

We have listed out the advantages of functions as

- ✓ Reusability of code leading to saving of memory space and reduction in code size.

While this is true, we also know that call statement to a function makes a compiler to jump to the functions and also to jump back to the instruction following the call statement. This forces the compiler to maintain overheads like STACKS that would save certain special instructions pertaining to function call, return and its arguments. This reduces the speed of program execution. Hence under certain situations specially, when the functions are small (fewer number of instructions), the compiler replaces the function call statement by its

definition ie., its code during program execution. This feature is called as inlining of a function technically called as **inline** function.

> An **inline** looks like a normal function in the source file but inserts the function's code directly into the calling program.
> Inline functions execute faster but require more memory space.

Now look at the following example.

```
// Program - 4.16
// inline functions

# include <iostream.h>
# include <conio.h>

inline float convert_feet(int x)
{
  return x * 12;
}

void main()
{
  clrscr();
  int inches = 45;
  cout << convert_feet(inches);
  getch();
}
```

```
// working of Program - 4.16
// inline functions

# include <iostream.h>
# include <conio.h>


void main()
{
  clrscr();
  int inches = 45;
  cout << inches * 12 ;
  getch();
}
```

As shown in the above example, the call statement to the function (convert_feet(inches) will be replaced by the expression in the return statement (inches * 12).

To make a function inline, one has to insert the keyword **inline** in the function header as shown in Program 4.16.

Note :

inline keyword is just a request to the compiler . Sometimes the compiler will ignore the request and treat it as a normal function and vice versa.

4.7 Scope Rules of Variables

Scope refers to the accessibility of a variable. There are four types of scopes in C++. They are:

1. Local scope
2. Function scope
3. File scope
4. Class scope

4.7.1 Local scope

```
// Program - 4.17
// to demonstrate local variable
# include < iostream.h
# include <conio.h>
void main ( )
{
      int a, b ;
       a = 10;
       b = 20;
  if (a > b)
  {  int temp; // local to this if block
     temp = a;
     a = b;
     b = temp;
   }
   cout << '\n Descending order…';
   cout << '\n' <<a << '\n' <<b;
   getch ( );
}
```

- A local variable is defined within a block.
- The scope of a local variable is the block in which it is defined.
- A local variable cannot be accessed from outside the block of its declaration.

Program-4.18 demonstrates the scope of a local variable.

```
//Program - 4.18
# include <iostream.h>
# include <conio.h>
void main ( )
{    int a, b;
  a = 10
  b = 20;
  if (a > b)
 {      int temp;
      temp = a;
      a= b;
      b = temp;
 }
   cout << a << b << temp;
  getch ( );
}
```

On compilation, the compiler prompts an error message:
Error in line no.13
The variable temp is not accessible.
The life time of a local variable is the life time of a block in its state of execution.
Local variables die when its block execution is completed.

- Local variables are not known outside their own code block. A block of code begins and ends with curly braces { }.
- Local variables exist only while the block of code in which they are declared is executing.

A local variable is created upon entry into its block and destroyed upon exit.

Identify local variables, in the Program-4.19 and also mention their scope.

4.7.2 Function scope

The scope of variables declared within a function is extended to the function block, and all sub-blocks therein.

```
// Program - 4.19
# include <iostream.h>
void main ( )
{ int flag = 1; a = 100;
  while (flag)
  {
    int x = 200;
   if (a > x)
   {  int j;
     -
    }
else
  { int h;
     -
  }}}
```

Local variable	Scope
1. x	Accessible in while block, and if blocks Accessible only in if (a>x) { } block
2. j	Accessible only in else block
3. k	

The variable flag of Program – 4.19 is accessible in the function main ( ) only. It is accessible in all the sub-blocks therein - viz, while block & if block.

The life time of a function scope variable, is the life time of the function block. The scope of formal parameters is function scope.

4.7.3 File scope

A variable declared above all blocks and functions (precisely above main ( )) has the scope of a file. The scope of a file scope variable is the entire program. The life time of a file scope variable is the life time of a program.

```
// Program - 4.20
// To demonstrate the scope of a variable

// declared at file level

# include <iostream.h>
# include <conio.h>
int i = 10;
void fun ( )
{ cout << i; }
void main ( )
{
   cout << i;
    while (i)
     {   -
         -
          -
      }
}
```

4.7.4 Scope Operator

The scope operator reveals the hidden scope of a variable. Now look at the following program.

```
// Program - 4.21
# include <iostream.h>
# include <conio.h>

int num = 15;

void main()
{
  clrscr();
  int num = 5;
  num = num + ::num;
  cout << num << '\t' <<
++::num;
  getch();
}
```

Have you noticed the variable **num** is declared both at file scope level and function main() level? Have you noticed the reference **::num ?** :: is called as scope resolution operator. It is used to refer variables declared at file level. This is helpful only under situations where the local and file scope variables have the same name.

4.7.5 Class scope

This will be discussed in Chapter 6.

Exercises

1. Construct function prototypes for descriptions given below:

a) procedural-function ( )
- is a function that takes no arguments and has no return value.

Solution -

```
        void procedural - function (void);
  OR    void procedural - function ( ) ;
```

b) manipulative - function ( ) takes one argument of double type and returns int type.

Solution -

i. int manipulative - function (double); OR
ii. int manipulative - function (double d); OR
iii. manipulative - function (double)

c) fun-default ( ) takes two arguments, once with a default integer value, and the other float, has no return type

Solution -

```
void  fun-default (float, int num = 10);
```

d) return - reference - fun ( ) takes two int arguments and return reference to int type

Solution -
int & return - reference - fun (int &, int &);

e) multi-arguments ( ) that takes two arguments of float, where the 1st argument is P1 should not be modified, and the 2nd argument is of reference type. The function has no return type.

Solution -
void multi - arguments (float const pi, int & a);

2. Identify errors in the following function prototypes:

a) float average (a, b);
b) float prd (int a,b);
c) int default-arg (int a = 2, int b);
d) int fun (int, int, double = 3.14);
e) void strings (char [ ]);

3. Given the function

```
void line (int times, char ch)
 {  cout << '\n';
    for (int i = 1; i < = times; i ++)
      cout << ch;
     cout << '\n';
}
```

Write a main ( ) function that includes everything necessary to call this function.

4. Write the scope of all the variables mentioned in this program.

```
# include <iostream.h>
float a, b ; void f1 (char);
int main ( )
{ char ch;
    -
     -

{ int i = 0;
-
-
-
}
  }
 void f1 (char g)
{ short x, y ;
}  =
```

Solution:
a,b - file scope
ch – function scope
- main ( )
i – scope within its block

x,y,g –function scope - f1 function

4. Identify errors in the following programs:

a)

```
# include <iostream.h>

xyz (int m, int n)
{ int m = 10;
    n = m * n;
    return n;
}
 void main( )
 { cout << xyz (9,27) ;}
```

Solution:

The variable ‘m’ is declared with function block, which is not permitted.

b)
```
# include <iostream.h>

void xyz ( );
void main ( )

{ int x = xyz ( ) ; }

void xyz ( )

{ return '10' ; }
```

Solution:

Function declared as void type, cannot have a return statement, hence the function call cannot be part of an expression

c)
```
# include <iostream.h>

void counter (int & a)
{ ++ a;}

void main ( )
{counter (50); }
```

Solution:

The actual parameter cannot be passed in the form of a value, as the formal parameter is of reference type

5. What will be the output of the following programs?

a)
```
# include <iostream.h>
int val = 10;
divide (int) ;
void main ( )
{int val = 5;
val = divide (::val/val);
cout << :: val<<val;
}
   divide (int v)
{ return v/2;}
```

Solution: 101

b)
```
# include <iostream.h>
divide (int v)
{ return v / 10;}
        void main ( )
{int val = -1;
val = divide (400) = = 40;
cout << "\n Val." << val;
}
```

Solution: 1 - Working

i) divide (400) yields
a value 40
ii) divide (400) == 40 is
interpreted as
40 ==40 since the
condition is true
val gets 1

c)
```
# include <iostream.h>
int incre (int a)
{ return a++; }
void main ( )
{int x = 10; x = incre (x); cout << x;}
```

Solution: 10

d)
```
# include <iostream.h>
# include <iostream.h>
void line( )
{static int v = 5;
  int x = v - - ;
   while (x)
   {cout << ' * ' ; x — ;
   }
   cout << 'In';
}
void main ( )
{ clrscr ( );
    for (int i = 1; i < = 5; i ++
        line ( ) ;
    getch ( );
}
```

Solution:

```
* * * * *
* * * *
* * *
* *
```

e)
```
# include <iostream.h>
first (int i)
{ return i++; }
second (int x)
{ return x —; }
void main ( )
{ int val = 50;
    val = val * val/val
    val = second (val);
    val = first (val);
    cout << "\n Val: " << val;
}
```

Solution:

Val : 50

6. Program writing….

a) Write a program in C++ to define a function called float cube (int, int, int). Write main ( ) function, to test the working of cube ( ).

b) Define a function unsigned long it factorial (int);

The factorial of a number is calculated as follows: For example Factorial of 5 is calculated as 1 x 2 x 3 x 4 x 5
Write a main ( ) function to calculate the factorial (n).

c) Define a function called as
char odd - even - check (int);

The function should return 'E' if the given number is even, otherwise 'O'. Write a main ( ) to test and execute the function odd-even-check (int) and also print relevant message.

d) Define a function int prime (int);

The function should return a value 1, if the given number is prime, otherwise -1. Write a main ( ) to test and execute the function and also print relevant message.