ARRAYS

An array is a group of variables of the same type. These variables are grouped together under a common name. Each individual variable is called as an array element

Array Declaration

```
Syntax
     Type var_name[size];
```

Here, type declares the base type of the array, which is the type of each element in the array, and size defines how many elements the array will hold.

```
Example:
    int a[10];
```

Array Initialization

```
Syntax:
    Type array_name[size]={value list};
```

Where type is the nature of data elements such as integer, floating, or character etc.; the array name is used to declare the name of the array; and the elements are placed one after other within the braces and finally ends with the semicolon. The first value is placed in the first position of the array, the second value in the second position, and son on.

Program:

Program to find the highest number in a single dimensional array of order n.

```
#include<iostream.h>
#include<conio.h>
void main()
{
 int a[100],n,i,big;
 clrscr();
 cout<<"Enter the no of elements";
 cin>>n;
 for(i=0;i<n;i++)
 cin>>a[i];
 big=a[0];
 for(i=1;i<n;i++)
 {
  if(a[i]>big)
   big=a[i];
 }
 cout<<"Big=”<<big;
 getch();
}

OUTPUT
Enter the no of elements 5
67
89
34
90
68
Big=90
```

7. FUNCTION

A function groups a number of program statements into a single unit and gives it a name. This unit can be called from other part of the program. The purpose of using function is to reduce the size of the program. There are times when some type of operation or calculation is repeated at many points throughout a program. For instance, we might use the factorial of a number at several points in the program. In such situations, we may repeat the program statements whenever they are needed. Another approach is to design a function that can called and used whenever required. This saves both time and space.

Declaration of functions

The function declaration gives the instruction to the compiler that the function body comes later.

Function Definition

```
Return_type function_name(argument list)
{
  Local variable declaration
  Executable part
  Return(expression)
}
```

Return_type:

Return_type specifies the data type of the value returned by the function, when it is invoked.

Function name:

Function name is a unique name identifying the function. The rules for writing a function name are the same as those for variable.

Argument list:

Argument list is a list of one or more arguments that are used by the function as an input for processing.

Return statement:

The keyword return is used to terminate function and return a value to its caller. The return statement may also be used to exit a function without returning a value. Return statement may or may not include an expression.

Calling a Function

Syntax:

```
function_name(arg,arg……);
```

Where, function_name is the name of the function as declared in the function definition.

arg1,arg2,…… are arguments passed to the function. These arguments must match both in number and type with the arguments listed given in the function definition. There are four types of functions in C++

1. **Function without argument and without return type**
2. **Function with argument and without return type**
3. **Function without argument and with return type**
4. **Function with argument and with return type**

1. Function without argument and without return type

```
#include<iostream.h>
#include<conio.h>
void power(void);
void main()
{
 clrscr();
 power();
 getch();
}
void power(void)
{
 int p=1,x,y,i;
 cin>>x>>y;
 for(i=1;i<=y;i++)
 {
  p=p*x;
 }
 cout<<p;
}

output:
5
3
125
```

2. Function with argument and without return type

```
#include<iostream.h>
#include<conio.h>
void power(int x,int y);
void main()
{
 int a,b;
 clrscr();
 cin>>a>>b;
 power(a,b);
 getch();
}
void power(int x,int y)
{
 int p=1,i;
 for(i=1;i<=y;i++)
 {
  p=p*x;
 }
```

```
 cout<<p;
}

output:
5
3
125
```

3. Function without argument and with return type

```
#include<iostream.h>
#include<conio.h>
int power(void);
void main()
{
 clrscr();
 cout<<power();
 getch();
}
int power(void)
{
 int x,y,p=1,i;
 cin>>x>>y;
 for(i=1;i<=y;i++)
 {
  p=p*x;
 }
 return(p);
}

output:
5
3
125
```

4. Function with argument and with return type

```
#include<iostream.h>
#include<conio.h>
int power(int x,int y);
void main()
{
 int a,b;
 clrscr();
 cin>>a>>b;
 cout<<power(a,b);
 getch();
}
int power(int x,int y)
{
 int p=1,i;
 for(i=1;i<=y;i++)
 {
  p=p*x;
 }
 return(p);
}
```

```
output:
5
3
125
```

Nested Function

A function may call one more functions and so on. There is no restriction in C++ for calling the number of functions in a program. It is advisable to break a complex problem into smaller and easily manageable parts and then define a function. Control will be transferred from the calling portion of a program to the called function block. If the called function is executed successfully, then control will be transferred back to the calling portion of a program.

Program:

```
#include<iostream.h>
#include<conio.h>
int ratio(int l,int m,int n);
int diff(int p,int q);
main()
{
 int a,b,c,r;
 clrscr();
 cin>>a>>b>>c;
 r=ratio(a,b,c);
 cout<<r;
 getch();
}
int ratio(int l,int m,int n)
{
 if(diff(m,n))
 {
  return(l/(m-n));
 }
 else
 {
  return(0);
 }
}
int diff(int p,int q)
{
 if(p==q)
  return(0);
 else
  return(1);
}

OUTPUT
7 5 5
0
```

Recursive Function

A function, which calls itself directly or indirectly again and again, is known as the recursive function.

Program:

```
#include<iostream.h>
#include<conio.h>
main()
{
 int n,k;
 clrscr();
 cout<<"Enter the value for n";
 cin>>n;
 k=fact(n);
 cout<<"Factorial of n="<<k;
 getch();
}
int fact(int p)
{
 int f;
 if(p==0)
  return(1);
 else
  f=p*fact(p-1);
 return(f);
}

OUTPUT
Enter the value for n 5
Factorial of n=120
```

Actual Arguments

An Actual argument is a variable of expression contained in a function this arguments pass save information to the function, and received by the formal arguments;

Formal Arguments

The arguments present in the function definition are known as formal arguments they also called as dummy arguments or the parametric variables. When the function is called these arguments will received the value from the actual arguments.

Call by value

Whenever a portion of the program invokes a function with formal arguments, control will be transferred from the main to the calling function and the value of the actual argument is copied to the function. Within the function, the actual value copied from the calling portion of the portion of the program may be altered or changed. When the control is transferred back from the function to the calling portion of the program, the altered values are not transferred back. This type of passing formal arguments to a function is technically known as call by value.

Program:

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

```
void swap(int p,int q);
void main()
{
 int a,b;
 clrscr();
 cin>>a>>b;
 cout<<a<<"\t"<<b<<endl;
 swap(a,b);
 cout<<a<<"\t"<<b<<endl;
 getch();
}
void swap(int p,int q)
{
 int temp;
 temp=p;
 p=q;
 q=temp;
}

OUTPUT
      6
      7
            6     7
            6     7
```

Call by Reference

Provision of the reference variables in c++ permits us to parameters to the functions by reference. When we pass arguments by reference, the 'Formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data.

Program:

```
#include<iostream.h>
#include<conio.h>
void swap(int &p,int &q);
void main()
{
 int a,b;
 clrscr();
 cin>>a>>b;
 cout<<a<<"\t"<<b<<endl;
 swap(a,b);
 cout<<a<<"\t"<<b<<endl;
 getch();
}
void swap(int &p,int &q)
{
 int temp;
 temp=p;
 p=q;
 q=temp;
}

OUTPUT
4
```

```
5
            4      5
            5      4
```

Function Overloading

An overloading function appears to perform different activities depending on the kind of data send to it. Function overloading is the logical method of calling several functions with the same name each redefinition of a function must use different type of parameters, or different sequence of parameters or different number of parameters.

Program:

```
#include<iostream.h>
#include<conio.h>
int sum(int a,int b);
int sum(int a,int b,int c);
double sum(double a,double b);
double sum(int a,double b);
double sum(double a,int b);
void main()
{
 clrscr();
 cout<<sum(5,4)<<endl;
 cout<<sum(6,5,3)<<endl;
 cout<<sum(4.5,5.6)<<endl;
 cout<<sum(4,6.5)<<endl;
 cout<<sum(6.4,3)<<endl;
 getch();
}
int sum(int a,int b)
{
 return(a+b);
}
int sum(int a,int b,int c)
{
 return(a+b+c);
}
double sum(double a,double b)
{
 return(a+b);
}
double sum(int a,double b)
{
 return(a+b);
}
double sum(double a,int b)
{
 return(a+b);
}

output:
9
14
10.1
10.5
9.4
```

Default Arguments

One of the most useful facilities available in c++ is the facility to define default argument value for functions. The default values are given in the function prototype declaration. Whenever a call is made to a function without specifying an argument the program will automatically assign values to the parameter from the default function prototype declaration.

Program:

```
#include<iostream.h>
#include<conio.h>
int sum(int a=45,int b=10,int c=20);
void main()
{
 clrscr();
 cout<<sum()<<endl;
 cout<<sum(5)<<endl;
 cout<<sum(4,5)<<endl;
 cout<<sum(6,3,8)<<endl;
 getch();
}
int sum(int a,int b,int c)
{
 return(a+b+c);
}
output:
75
35
29
17
```

Inline Function

Functions are used to save the memory space. When the complier sees a function call, it normally generates a jump to the function. At the end of the function it jumps back to the instruction following the call. Even though the function saves the memory space it takes extra time to execute. To avoid this flow down processing of the program inline function is used. When a function declared as a inline, the actual coding is inserted in the place of function call while compiling.

To make a function as inline, the key word inline must be added before the function definition. While creating a inline function there is no need for function prototyping.

Program:

```
#include<iostream.h>
#include<conio.h>
inline int sum(int a,int b)
{
 return(a+b);
}
void main()
{
 clrscr();
```

```
 cout<<sum(5,6)<<endl;
 getch();
}
output:
11
```