UNIT – II

Function and Program Structures: Introduction, Defining a Function, Return Statement, Types of Functions, Actual & Formal Arguments, Local & Global Variables, Default Arguments, Structure of C++ Program, Order of the Function Declaration, Manually invocated Functions, Nested Functions, Scope Rules, Side Effects, Storage Class Specifiers, Recursive Function, Pre-processors, Header Files, Standard Functions.

Arrays: Introduction, Array Notation, Array Declaration, Array Initialization, Processing with Arrays, Arrays and Functions, Multidimensional Array, Character Array.

Pointers and Strings: Introduction, Pointer Arithmetic, Pointers and Functions, Pointers to Functions, Pointers and Arrays, Array of Pointers, Pointers to Pointers, Pointer and Strings, Deciphering Complex Declarations.

Structures, Unions and Bit Fields: Introduction, Declaration of Structure, Processing with Structures, Initialization of Structures, Functions and Structures, Array of Structure, Array within a Structure, Nested Structure, Pointer and Structure, Unions, Bit Fields, Typedef, Enumerations.

❖ Function:

In real world the program generally extend to thousands of lines of code that make their maintenance and readability a complex task. To make these program easy to manage and easy to read, functions are used. Functions are the building blocks of structured programming approach. They are self contained and well defined named group of statements that are aimed at accomplishing a specific task or action in the program.

A function is used to combined a particular set of instructions that needs to be accessed repeatedly in a program. Hence, the use of functions prevents the repetition of code and also reduces the amount of memory required to store the entire program. This is because a function's code is stored once in memory and can be executed whenever required.

❖ Function Declaration:

Like variables, functions also need to be declared before they are used in programs. A function declaration is also known as **function prototype.** Function prototype is model or a blueprint for a function that informs the C++ compiler about the return type, the function name, and the number and data type of the arguments passed to the function. Function name together with parameter list is known as **function signature** and it does not include return type of a function.

Syntax,

return_type function_name (parameter_list);

Where,

return_type	= the data type of the value returned by the function (by default it is int)
function_name	= the name of the function
(parameter_list)	= the list of variables along with their data types

❖ Function Definition

In order to use a function in a program, the function must be first defined somewhere in a program. A **function definition** contains the code that specifies the actions to be performed.

Syntax,

```
return_type function_name (parameter_list)      //function header
{                                               //function body is always
        statement 1;                            // specified
        statement 2;                            //within curly braces
          .
          .
        Statement N;
}
```

In C++, the function definition consists of two parts, namely, the function header and the function body. The first line of the function definition that specifies the *return_type, function_name* and *parameter_list* of the function is known as the **function header,** and the sequence of statements that are executed when the function is invoked is known as the **function body.**

The definition of the function can appear either before main( ) or after main ( ). In case, the function definition appears before main ( ), there is no need for declaring the function as the function definition itself acts as the function declaration. However, if function definition appears after main( ), the function needs to be declared explicitly in the program.

The concept of function definition before main ( )

```
#include<iostream.h>
int func (int x, float y)
{
        //function body
}
int main ( )
{
        //body of main
}
```

The concept of function definition after main ( )

```
#include<iostream.h>
int func(int,float);
int main ( )
{
        //body of main
}
int func (int x, float y)          //function definition
{
        //body of function
}
```

❖ Invoking Function

The function that calls another function is known as the **calling function** and the function that is being called is known as the called function. when a function is called, the control immediately passes from the calling function to the called function. the called function then executes its body after which the control returns to the next statement in the calling function.

The calling function passes information to the called function through **arguments.** The arguments that appear in the function call are known as **actual arguments** and the arguments that appear in the function definition are known as **formal arguments.** The number of actual arguments, their order and type in function call must match with that of formal arguments. For example:

```
#include<iostream.h>
void sum(int, int)
int main ( )
{
        int a, b;
        cout<< "Enter First Number:-";
        cin>>a;
        cout<< "Enter Second Number:-";
        cin>>b;
        sum(a,b);                          //actual arguments
}
```

```
void sum (int x, int y)    //formal argumnets
{
        Cout<<"The Sum is : "<< ( x + y);
}
```

```
Output:
Enter First Number      : 15
Enter Second Number     : 16
The Sum : 31
```

❖ Default Arguments

Whenever a function is called, the calling function must provide all the arguments specified in the function's declaration. If the calling function does not provide the required arguments, the compiler raises an error. However, C++ allows a function to be called without specifying all of its arguments. This can be achieved by assigning a default value to the argument. default value is specified in the function declaration and is used when the value of that argument is not passed while calling the function.

If a function call does not specify an argument, the default value is passed as an argument to the function. in case a function call specifies an argument, the default value is overridden and the specified value is passed to the function. For example,

```
#include<iostream.h>
int addition(int a, int b, int c=15);
{
        int x;
        x = addition (5,10);        //default argument missing
        cout<<"addition with default value= "<<x;
        x = addition(5,10,20)     //default argument override
        cout<<"addition with override value="<<x;
}
int addition(int a, int b, int c)
{
        int z;
        z=a+b+c;
        return (z);
}
```

❖ Constant Argument

When an argument is passed to a function, its value can be modified within the function body. If the user wants the value of an argument to be intact throughout the function then such argument must be declared as constant. As the value of constant arguments cannot be changed in the called function, an attempt to modify such argument results in compile-time error. An argument to a function can be made constant by prefixing const keyword to the data type of the argument.

❖ Structure as Argument

Like ordinary variables, structure variables can also be passed by value to a function.

```
#include<iostream.h>
struct details
{
        float qty;
        float amt;
        float total;
};
void calc(details);
```

```
int main ( )
{
        details  d1=(5,10,0);
        calc(d1);
        cout<<”Value of Total in main = “<<d1.total;
        getch();
}
void calc(details d2)
{
        d2.total = d2.qty * d2.amt;
        cout<<”Value of Total in function = “<<d2.total;
}
```

❖ Array as Argument

Since an array represents a complete block of memory, an entire array cannot be passed to a function. Hence, a reference to the array is passed to the function to avoid copying huge amount of data.

Though an array is passed as a reference, a reference operator or ampersand (&) is not used to denote a reference argument because the array name itself contains the address of the first element.
The Syntax,

return_type function_name (type [ ]);

Syntax for a function call passing an array as an argument is:

function_name (array_name);

❖ Call by Value:

When a function is called by value, the values of the actual arguments are copied into the formal arguments and the function works with these copied values. As a result, any changes made to the copied value in the called function do not affect the value of the actual argument in the calling function.

By default, functions are called by value in C++. The call by value method is useful when an argument is to be used only for passing information to a function and does not need to modify the original values.

❖ Call by Reference

In addition to call by value, function can also be called by reference. C++ provides two ways of passing arguments as reference to a function. the argument can be passed either as a reference type or as a pointer type.

When an argument is passed by reference, an alias or a reference of the actual argument is passed to the formal argument. unlike call by value method, call by reference does not create a copy of the actual argument and the called function works with the original values. This implies that any changes made to the variable in the function body are reflected in the calling function.

In call by reference, a reference to an argument is created by suffixing the reference operator or ampersand (&) to the data type of an argument in both the function definition and the declaration .

Arrays: Introduction, Array Notation, Array Declaration, Array Initialization, Processing with Arrays, Arrays and Functions, Multidimensional Array, Character Array.

Pointers and Strings: Introduction, Pointer Arithmetic, Pointers and Functions, Pointers to Functions, Pointers and Arrays, Array of Pointers, Pointers to Pointers, Pointer and Strings, Deciphering Complex Declarations.

Structures, Unions and Bit Fields: Introduction, Declaration of Structure, Processing with Structures, Initialization of Structures, Functions and Structures, Array of Structure, Array within a Structure, Nested Structure, Pointer and Structure, Unions, Bit Fields, Typedef, Enumerations.

❖ Array

C++ uses variables of different built-in data types to store data. However, these variables are incapable of holding more than value of a time. For example, a single variable cannot be used for storing the names of all the students in a class. For such purposes, C++ provides a different kind of data type, called arrays.

Arrays are defined as a fixed size sequence of same type of data elements. These data elements can be of any built-in or user defined data type. The elements of an array are stored in contiguous memory locations and each individual element can be accessed using one or more indices or subscripts. A subscript or an index is a positive integer value, which indicates the position of an element in an array. Arrays are used in situations where a programmer wants to store a list of data items into a single variable and also wants to access and manipulate individual elements of the list. Arrays can be either single dimensional or multi dimensional depending upon the number of subscripts used.

❖ Single-Dimensional Array

A single dimensional array is the simplest form of an array that requires only one subscript to access an array element. Like an ordinary variable, an array must have been declared before it is used in the program.

Syntax,

```
Data_type       array_name [size];
```

Example,

```
int marks[5];
```

The size of an array should always be enclosed in square brackets [ ] and must be a positive and constant integer value.

Initialization of Single Dimensional Array

Once an array is declared, the next step is to initialize each array element with a valid and appropriate value. An array can be initialized at the time of its declaration.

Syntax,

```
data_type array_name[size] = {value1, value2, …………value n};
```

The values are assigned to the array elements in the order in which they are listed. For example,

```
int marks[5] = {58,75,69,58,80};
```

If an array is declared and initialized simultaneously, they specifying its size is optional

```
int marks[ ] = {58,75,69,50,80};  is also valid.
```

Accessing Single-Dimensional Array elements

Once an array is declared and initialized, the values stored in the array can be accessed any time. Each individual array element can be accessed using the name of the array and the subscript value. Every element in an array is associated with a unique subscript value, starting 0 to Size-1.

The memory location, where the first element of an array is stored, is known as the **base address,** which is generally referred to by the name of the array.

Single dimensional arrays are always allocated contiguous blocks of memory. This implies that all the elements in an array stored next to each other.

The Memory representation of Array

Marks[0]	Marks[1]	Marks[2]	Marks[3]	Marks[4]
58	75	69	50	80
2001	2003	2005	2007	2009

Manipulation of Single Dimensional Array Element

Once the array elements are accessed, a number of operations can be performed on them. These operations include finding the sum, average, maximum, or minimum, sorting or searching of the array elements. etc.

Strings-Array of Characters

In addition to integer and floating point elements, an array can have characters as its elements. an array containing a group of characters is known as a character array. For example,

char name[5]={'a', 'b', 'c', 'd', 'e'}

An array of character, in which the last character is always a null character, ('\0') representing a string. Thus, while declaring a character array that holds a string, the size is always declared one more than the number of characters in the string.

❖ Multi-Dimensional Array

Multi-Dimensional Array can be described as an 'array of arrays' that is each element of the array is in itself an array. A multi dimensional array of dimension is a collection of items that are accessed with the help of n subscripts.

Generally, the arrays of three or more dimensions are not used because of their huge memory requirements and the complexities involved in their manipulation.

Two Dimensional Array

A two dimensional array is the simplest form of a multi dimensional array that requires tow subscript values to access an array element. Two dimensional arrays are useful when data ebign processed can be arranged in the form of rows and columns.

Syntax,

```
data_type array_name [row_size] [column_size];
```

Example,

```
int a[3][2];
```

Initialization of two dimensional array

Like a single dimensional array, a two dimensional array can also be declared and initialized at the same time. To understand the concept of two dimensional array initialization, consider this statement,

```
int a[3][2] ={    {101,51},
                  {102,67},
                  {103,76}        };
```

In this statement, the first two element are stored in the first row, the next two in the second and so on.

Accessing two dimensional array elements

Once a two dimensional array is declared and initialized, its elements can be accessed using two subscripts. The syntax,

```
array_name [row] [column];
```

The first subscript (row) specifies the row number and the second subscript (column) specifies the column number. Both subscripts together specify the position of the element within array.

Matrix and memory representation of two dimensional array.

int a[3][2];

a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]
101	51	102	67	103	76
2001	2003	2005	2007	2009	2011

first row second row third row

101	51
102	67
103	76

Array of Strings

An array of string is simply a two dimensional array of characters where each row represents a string. An array of string is declared and initialized in the same way as any two dimensional array of type int, float, etc. To understand the concept of declaring and initializing an array of strings, consider this statement,

Example,

```
char names[3][8]={     "Nikhil",
                       "Raman",
                       "Nandita"      };
```

	0	1	2	3	4	5	6	7	
2001	'N'	'i'	'k'	'h'	'i'	'l'	'\0'		names[0]
2009	'R'	'a'	'm'	'a'	'n'	'\0'			names[1]
2017	'N'	'a'	'n'	'd'	'i'	't'	'a'	'\0'	names[2]

Pointers

A pointer is a variable that contains the address of another variable. In other words, a pointer 'point to' a particular location in the memory. The memory can be allocated and de-allocated to the variables at run time using pointers.

❖ Declaring and initializing pointers

Like other variable, a pointer variable must be declared before it is used in the program. The declaration of pointers is similar to that of an ordinary variable that is, by specifying the data type and the name of the variable. However, the only difference is that the data type of a pointer is followed by an asterisk '*'

Syntax,

```
data_type *pointer_name;
```

The data type of a pointer variable informs the compiler about the type of variables the pointer can point to.

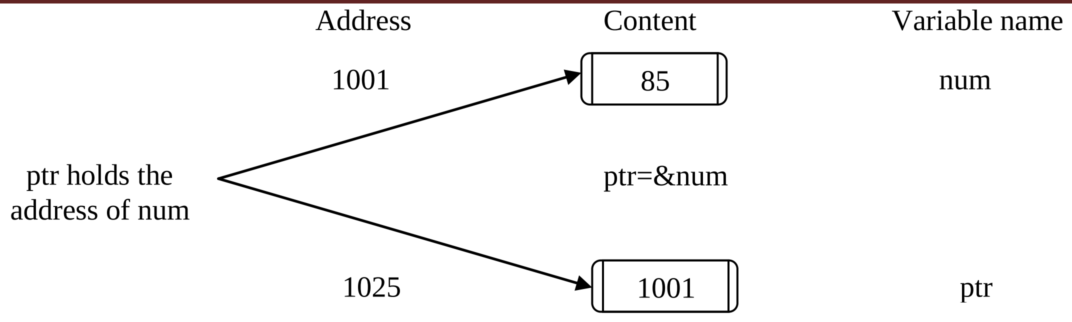
Example,

```
int * iptr;
float *fptr;
```

Once a pointer variable is declared, it needs to be initialized with the memory address of a variable of an appropriate data type. The address of a variable can be obtained by prefixing its name with the pointer operator named address of operator. The address-of (&) operator also called reference operator is a unary operator that returns the address of a variable stored in the memory.

For example,

```
int num=85;
int *ptr;
ptr=&num;
```



❖ **The Null pointer**

A pointer must be initialized with 0, NULL or an address of corresponding data type, when it is declared or in an assignment. A pointer set to 0 or Null is called **NULL pointer**. Any pointer can take a null value to represent that the pointer is pointing to "nowhere"

For example,

```
int *iptr=NULL;
```

❖ **Dereferencing Pointer**

Once a pointer variable is initialized, the contents of the variable pointed to by it, can be accessed with the help of another pointer operator called indirection or dereference operator. The indirection or dereference (*) operator is a unary operator used with a pointer variable to 'indirectly' access the contents of the memory location pointed to by the pointer. In other words, indirection (*) operator is the complement of &. It means the value stored at the address pointed to by the pointer.

The syntax,

```
*pointer_name;
```

Using pointers, the contents of the variable can not only be accessed but can be modified also.