COMPUTER PROGRAMMING & NUMERICAL METHODS
INTRODUCTION TO CLASS AND OBJECTS

MODULE - I

Introduction to Computer programming concept - Algorithm and flow chart, Basics of procedure oriented and object oriented programming. Introduction to C++: Structure of C++ program; Key words; Identifiers; Data types – integer, real, character, string, Boolean, enumeration, array and pointer; Constant and Variables; Escape sequences; Operators – assignment, arithmetic, relational, logical, increment & decrement, conditional, size of, comma and bitwise operators; Statements – simple & compound, declaration statements, Control statements -if, if-else, switch, for loop, while, do-while, break and continue statements, Input and output streams, Arrays – one dimensional & two dimensional; Functions- inline functions, function over loading, Functions with default arguments, recursion, pointers. Simple programs using above features.

MODULE -II

Introduction to Class and Object- definition, data members, member function, private & public member function, member access, friend declaration, class objects, predefined classes, initialization, constructor and destructor; Operator overloading, Inheritance- base class and derived class; Input/output stream library - ifstream, ofstream , fstream, class flies. Simple problems using the above features.

MODULE-III


References:

1. Ashok M. Kamthane, Object oriented Programming with ANSI & Turbo C++, Pearson Education.
3. Stanley B. Lippman and Josee Lajoie, C++ Primer, Pearson Education.

MODULE-1
INTRODUCTION TO PROGRAMMING

Computer is a machine which cannot perform of its own and it needs to be instructed. Let us consider a simple task of addition of two numbers. For that first we need to input the values for A and B, after processing the addition the sum should be stored in C.

So we have to give instructions, order of processing and data required for instructions. And these set of instructions is known as computer program.

“A computer program is a set of instructions that instructs a computer how to perform a task”.

To develop a program

1. Define the problem
2. Develop algorithm
3. Test algorithm
4. Coding
5. Test for errors
6. Implementation
7. Maintenance & Enhancement
Algorithm is defined as a finite sequence of explicit instructions that when provided with a set of input values produces an output and then terminates.

For example

Step 1: Start
Step 2: Read two numbers A and B
Step 3: Add A and B
Step 4: Store it in C
Step 5: Display C
Step 6: Stop

A Flow chart is a pictorial representation of algorithm in which the steps are drawn in the form of different shapes of boxes and logical flow is indicated by arrows. The primary purpose of flowchart is to help programmer in understanding the logic of program and outlines the general procedure.
PROGRAM PARADIGM

There is a program paradigm which refers how a program is written in order to solve a problem. The important program paradigms are

Procedural Programming

*Decide which procedures you want;*

*Use the best algorithms you can find.*

The focus is on the processing the algorithm needed to perform the desired computation. Languages support this paradigm by providing facilities for passing arguments to functions and returning values from functions.

Modular Programming

Over the years, the emphasis in the design of programs has shifted from the design of procedures and toward the organization of data. A set of related procedures with the data they manipulate is often called a module. The programming paradigm becomes:

*Decide which modules you want;*

*Partition the program so that data is hidden within modules.*

This paradigm is also known as the data hiding principle. Where there is no grouping of procedures with related data, the procedural programming style suffices.
Object Oriented Programming

The major motivating factor in the invention of object-oriented approach was to overcome some of the problems encountered in the procedural approach. OOP treats data as a critical element in programme development. It ties data more closely to the functions that operate on it and does not allow it to flow freely around the system. OOP allows us to decompose a problem into a number of entities called objects.
OBJECT ORIENTED PROGRAMMING (OOP)

The major motivating factor in the invention of object-oriented approach was to overcome some of the problems encountered in the procedural approach. OOP treats data as a critical element in programme development. It ties data more closely to the functions that operate on it and does not allow it to flow freely around the system. OOP allows us to decompose a problem into a number of entities called objects.

Features of OOP are:

- Emphasis is on data rather than procedure.
- Programs are divided into objects.
- Data structures are designed such that they characterize the objects.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate to each other through functions.
- Follows bottom-up approach in program design.

CONCEPTS OF OBJECT ORIENTED PROGRAMMING

1. Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, an address, a table of data or anything. Each object contains data and a code or function to manipulate data. Therefore, data and functions are binded together to form objects.
2. Classes

We can make the above set of data and code in an object into a user-defined data type with the help of a class. Once we have created a class, then we can make any number of objects belonging to that class. This means objects are variables of type class. *A class is thus a collection of objects of similar type.* For example, if we make a class fruit, then mango, apple, orange are all objects of class fruit.

3. Data Abstraction and Encapsulation

*When we wrap up the data and functions into one unit, called class, it is called encapsulation.* Encapsulated data is not available to the outside world and only functions belonging to that class can access it. These functions provide the interface between the objects data and the program. This insulation of data from direct access by the program is called *data hiding.*

*Abstraction as the act of representing essential features without including the background details or explanations.* Classes use the concept of data abstraction and are defined as a list of abstract attributes.

4. Inheritance

*Inheritance is a process by which objects of one class acquire the properties of the objects of another class.* It supports the concept of hierarchical classification. For example
In OOP, the concept of inheritance provides the idea of reusability. This means that we can add some more features to an existing class without modifying it. We can do this by deriving a new class from two existing classes. The new class will have some features from each class.

5. **Polymorphism**

*Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behavior in different instances.* Let us consider the example of the addition operation. If two numbers are added the operation produces the sum. If we add two strings (set of characters) then the operation produces a group string.

\[ 2 + 3 \Rightarrow 5 \]

\[ "The" + "House" \Rightarrow "The House" \]

This means that we can use a single function name to handle different numbers and different types of arguments.
6. **Dynamic Binding**

Binding is the linking of a procedure call to the code to be executed in response to the call. *Dynamic binding means that we do not know the code associated with the given procedure call until the time of call at run-time.* It is associated with polymorphism and inheritance. For example, if we define the same function name for more than one purposes, the exact nature of call is not known until the call has actually been made at run-time. Consider the addition operation; until the call is actually made at run-time, the procedure does not know whether it has to add two numbers or two strings. Hence the procedure gets bound dynamically at run-time.

7. **Message Communication**

An object-oriented program consists of a set of objects that communicate with each other. Therefore, the process of programming in an object-oriented language involves the following steps:

(a) Creating class

(b) Creating objects from class definition

(c) Establishing communication between objects.

Objects communicate with one another by sending and receiving information in the same way as we pass messages to each other. A message for an object is a request for execution of a procedure. Message passing involves specifying the name of the object, name of the function (message) and the information to be sent.
INTRODUCTION TO C++

C++ is an object-oriented programming language which was originally called ‘C with classes’. *It was developed by Bjarne Stroustrup at AT&T Bell Laboratories* in the early eighties. Stroustrup admired Simula 67 and he strongly supported C, so he combined the features of the two languages into C++. As a result, C++ supports object-oriented programming features more powerfully and still retains the power and elegance of C language.

The three most important facilities that C++ adds on to C are classes, function overloading and operator overloading. These features enable us to create abstract data types, inherit properties from existing data types and support polymorphism.

C++ is a versatile language for handling large programs. We can use it for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems. C++ applications are easily maintainable and expandable.

A SIMPLE C++ PROGRAM / STRUCTURE OF C++ PROGRAM

```cpp
#include <iostream.h>

void main ( )
{
    cout <<"C++ is better than C.\n";
}
return0;
```
For input/output in C++ we have to include `iostream.h` header file. C++ programs are also a collection of functions. The above example consists of only one function `main()`. C++ statements should terminate with semicolons. `Cout` is an output operator. `Cout` is a predefined object that represents the standard output stream in C++, here the screen. The operator `<<` is called the insertion or put to operator. It gives output on the screen. The corresponding command for inputting values from screen to the program is `cin`.

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello World!"
    return 0;
}
```

### Structure of a C++ program

<table>
<thead>
<tr>
<th>Include Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class declaration</td>
</tr>
<tr>
<td>Member functions definitions</td>
</tr>
<tr>
<td>Main function program</td>
</tr>
</tbody>
</table>

### KEYWORDS AND IDENTIFIERS

**Keywords are known as reserved words.** Each keyword has a specific meaning to the C++ Compiler. Ex: `int`, `cout`, `goto`.

**Identifiers are user defined words** and refer to the names of variables, functions, classes etc. Example: `sum`, `showdata`.

“**A keyword has same meaning even if it is used in different programs but identifiers have different meanings**”.
For example ‘p’ is principal amount in one type program and taken as pressure in another program.

**Rules for constructing identifiers**

- An identifier can have maximum 31 characters.
- The permissible characters in an identifier include letters, digits and an optional underscore symbol.
- It should start with a letter.
- It should not be a Keyword.

**DATA TYPES**

In C++, there are three kinds of data types

- Built-in data types
- User-defined data types
- Derived data types
Built-in Data Types

It is mandatory to declare the type of data. When a variable is declared as of a particular type it gets memory allocation accordingly. For instance the integer type variable takes 4 bytes or one word. Built in data types are also known as basic or fundamental data types. The various types are:

   a) Integer
      All numeric data items with no fractional part belong to integer type. The keyword int is used to represent integer type.

   b) Character
      All single characters used in programs belong to character type. Keyword used is Char.

   c) Real or Float
      All numeric data items with fractional part belong to real type. The keyword used is float.

   d) String
      The data type used for storing non-numerical values that are longer than one single character is known as string.

   e) Boolean
      It is used to declare a variable whose value will be set as true (1) or false (0). The keyword used is bool. It is used to check the state of a variable, an expression or a function as true or false.

   f) Void
      Void is also a built in data type which we use to represent functions that return no value. There are no void objects. There are no references to void.
User-defined Data Types

User defined data types are data types defined by the user using the basic data types. These are of various types:

(a) Structure

A structure is a heterogeneous user defined data type. A structure may contain different data types. A structure is a convenient tool for handling a group of logically related data items. Keyword used is `struct`.

(b) Class

A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.
(c) Union

Unions allow one same portion of memory to be accessed as different data types, since all of them are in fact the same location in memory. Its declaration and use is similar to the one of structures but its functionality is totally different.

(d) Enumeration

Enumeration is an user defined data type which provides a way for declare a new data type and define the variables of these data types that can hold. The keyword used is `enum`.

Derived Data Types

These are the data types derived from the basic or user-defined data types. They are

(a) Arrays

An array is a collection of data storage locations, each of which holds the same type of data. These storage areas are called elements. Arrays are used to handle large amounts of data.

(b) Pointers

A pointer stores addresses in C++. They provide much power and utility for the programmer to access and manipulate data in ways not seen in some other languages.
### CONSTANTS AND VARIABLES

**Constants are those in which its value does not change during the execution of the program enclosing it.** The constants may be Numerical constants, Real constants and Character constants.

**Variables are those in which the value can be changed during the execution of program containing it.** The name of variable should be a valid identifier and the name selected for the variables must reflect the purpose of variables. The variable can be declared anywhere in a program but before their usage.

### Scope of Variables

C++ provides us the facility to declare any variable anywhere in the program but not before it is used with the help of variables. The variable declared before the `main()` function known as *global variable* will be visible throughout the program while the variables declared in the block are known as *local variables*. The global variables will be lost as soon as the program ends because they will lose the memory allocated to them. Similarly, the local variable will be destroyed as soon as the scope of the block ends.

### Dynamic Initialisation of Variables

A variable can be initialized at run time using expressions at the place of declaration. For example

\[
\text{int } i = \text{strlen (string);} \\
\text{float area = length/breadth;} 
\]
The above two examples show that both the declaration and initialization of a variable can be done simultaneously at the place where the variable is used for the first time.

**Reference Variables**

C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alias for a previously defined variable. For example, if we make the variable `prod` a reference to the variable `mult`, then `mult` and `prod` can be used interchangeably. A reference variable is created as follows:

\[
\text{data type and reference-name} = \text{variable-name}
\]

Example

```c++
int mult = 50;
int & prod = mult;
```

**Manipulators**

Manipulators are used for manipulating and modifying data that are to be displayed on the command prompt. These manipulators are used with insertion operation “<<” for cout statement. Some of these manipulators are `endl` and `setw`.

- `endl` manipulator is used to insert on line in their stream to be displayed. It is similar to the ‘\ln’.
- `setw` manipulator is used for proper formatting of the output stream to be displayed. It is used to specify the field widths of the stream for
example; set (n) manipulator causes the value that follows it in the stream to be displayed within field n character wide.

### ESCAPE SEQUENCES

An escape sequence always begins with backslash and is followed by one or more characters. The commonly used escape sequences are

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>Bell or Beep</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
</tr>
<tr>
<td>\r</td>
<td>Return</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>'</td>
<td>Single quotation</td>
</tr>
</tbody>
</table>

The backslash is an escape from normal way of interpreting characters.

### OPERATORS

An operator is a symbol, which instructs the computer to perform the specified manipulation over some data.

The various operators in C++ are

- Arithmetic operators
- Assignment operators
- Relational operators
- Scope Resolution operators
- Bitwise Logical operators
- Increment-Decrement operators
- Memory Management operators
- Type Cast operator
- Conditional operator
- Comma operator
- Size of operator

**Arithmetic Operators**

The arithmetic operators are used to perform arithmetic operations like addition, subtraction, multiplication etc. over numeric values by constructing arithmetic expressions.

Operators can be divided into two categories – *Binary and Unary*. Addition, subtraction, multiplications, etc. are binary operators and applied on two operands. Increment and decrement are unary operands as they are applied on single operand. The important arithmetic operators are +,-, /, *, %, ++ and --.

**Assignment Operators**

The basic assignment operator ‘=’ is used to assign value to the variable. For example A = B; in which we assign value of B to A. Various assignment operators available in C++ are = , + = , − = , * = , / = , % = , & = , | = , | = , << = , >> =.

**Relational Operators**

Relational operators, also called comparison operators, are used to determine the relationship between two values in an expression. The return value will be either true or false. Logical operators allow you to combine
the results of multiple expressions to return a single value that evaluates to either true or false. The various relational operators are $>$, $\geq$, $<$, $\leq$, $=$, $!=$.

**Boolean Operators**

Boolean operators allow you to combine the results of multiple expressions to return a single value that evaluates to either true or false.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Use</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>A &amp;&amp; B</td>
<td>Conditional AND: If both A and B are true, result is true. If either A or B are false, the result is false. But if A is false, B will not be evaluated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bitwise Operators**

Computer data is represented as binary numbers (1’s and 0’s). The binary representation of the number 43 is 0101011. Bit position 0 (count from left to right hand side) is the least significant bit that value is 1 here, and in the above case bit 7 is the most significant. Bitwise operator allows manipulating integer variables a bit level.
| $\gg$ | $A \gg B$ | Right Shift : Shift bits of $A$ right by distance $B$, $0$ is introduced to the vacated most significant bits, and the vacated least significant bits are lost. The following shows the number 43-shifted right once ($42 \gg 1$):

|   |        | 0101011 | 43
|   |        | 0010101 | 21 |

| $\ll$ | $A \ll B$ | Left Shift : Shift bits of $A$ left by distance $B$, the most significant bits are lost as the number moves left, and the vacated least significant bits are 0.

| $\sim$ | $\sim B$ | Bitwise Component : The bitwise component negates the bits. If a bit is 1, it becomes 0, and if a bit is 0, it becomes 1. The following shows the complement of number 43:

|   |        | 0101011 | 43
|   |        | 1010100 | 84 |

| $\&$ | $A \& B$ | Bitwise AND : Bitwise AND is true only if both bits are set. Consider 23 & 12:

|   |        | 10111  | 23
|   |        | 01100  | 12
|   |        | 00100  | 4  |
Scope Resolution Operator

In C++ is a block structured language, which means that the same variable name can be used to have different time meanings in different blocks. The scope of the variable extends from the point of declaration to the end of the block. In C, we cannot access the global version of the variable from within the inner block. C++ has resolved this problem by introducing a new operator :: called the scope resolution operator.

Bitwise Logical Operators

Bitwise operators enable us to perform manipulations over data at bit level, thus it will in turn help us to perform lo level functions.

They are & -bitwise AND , | - bitwise OR , ^ - bitwise exclusive OR , ~ - bitwise NOT , >> - right shift , << - left shift
Increment-Decrement Operators

The operators ++ and − − are used for respectively incrementing and decrementing the value of the variable by 1.

Memory Management Operators

C++ defines two more operators new and delete. These are unary operators. New performs the task allocating the memory and delete performs the task of freeing the memory. Its general form is

\[
\text{Pointer-variable} = \text{new} \text{ data-type};
\]

For example

\[
P = \text{new} \text{ int};
\]

Here the new operator allocates sufficient memory to hold a data object of type int and returns the address of the object to pointer variable p. The statement delete p; will free the memory allocated to p.

Type Cast Operator

Generally arithmetic expressions use only one type of variables and constants. In C++ we use a set of rules to make type conversions automatically but the values are converted from lower to higher ranking. These rankings are given below from left to right in lower to higher ranking respectively:

\[
\text{Char} < \text{int} < \text{long} < \text{float} < \text{double} < \text{long double}.
\]
We can explicitly convert the type of a variable in C++ using the type cast operator. For example:

```cpp
int i, sum;

float average; average = float (sum)/i;
```

This will convert `sum` to floating point variable (which is an integer data type) and as a result the division operation will be performed as a floating point operation.

**Conditional Operator [?:]**

It helps in two-way decision making. The general form is

```
(Expression1)?(Expression2): (Expression 3);
```

Here expression 1 is evaluated and if it is true expression 2 becomes the value otherwise expression 3 becomes the value. Since three expressions are involved the operator is called *ternary operator*.

**Comma Operator**

It is basically used to link the two related expressions.

```
Expression1, Expression2;
```

The expressions are evaluated from left to right.

**Size of Operator**

The `sizeof()` operator is used to find the size of variable or the size of a data type in terms of number of bytes.
In each program all the statements from the first to the last gets executed without fail in a serial manner. That type of execution is known as sequential execution.

If the selection of some statements for execution depends on whether a condition is true or false then that type of execution is called conditional execution or selection.

A single statement is called a simple statement and set of statements enclosed within a pair of opening and closing braces is called compound statement.

**DECISION MAKING STATEMENTS**

**Simple if statement**

A simple if statement only allows selection of a statement when a condition occurs. If there are alternative statements, some which need to be executed when the condition holds, and some which are to be executed when the condition does not hold.

For example

```c
if (disc >= 0.0)
    cout << "Roots are real";
if (disc < 0.0)
    cout << "Roots are complex";
```
Here disc is a float variable and the first statement checks whether number is greater or equal to zero. If condition satisfies it prints Roots are real, and if the disc variables is less than zero then it prints Roots are complex.

This technique will work so long as the statements which are executed as a result of the first if-statement do not alter the conditions under which the second if-statement will be executed. C++ provides a direct means of expressing this selection.

**if-else Statement**

The if-else statement specifies statements to be executed for both possible logical values of the condition in an if statement. The general of the if-else statement is

```plaintext
if (condition)
    statement1
else
    statement2
```

If the condition is true then statement1 is executed, otherwise statement2 is executed. Both statement1 and statement2 may be single statement or compound statements. Single statements must be terminated with a semi-colon.

**CONTROL STATEMENTS**

**goto Statement**

The statement does not require any condition. It passes control anywhere in the program. The general format is shown below:
goto label ;
........
........
label ;

where, label is any valid label either before or after goto statement. Label is an identifier starting with character.

Switch Case

The switch statement can be used instead of the else if construct to implement a sequence of parallel alternatives. It act as a selector of one block out of many blocks of statements.

Syntax is

Switch (expression)
{case constant1: statementList1;
case constant 2 : statementList2;
case constant 3 : statementList3;
:
:
case constantN : statementListN;
default :statementList0 ;
}

LOOPS

for Loop

The for loop controls the repetition of one or more C++ statements. The statements in the for loop repeat continuously for a specific number of times, whereas the while and do-while loops repeat until a certain condition is met, the for loop repeats until a specific count is met.
Therefore, it is best to use a for loop when you know exactly how many iterations the loop will be making. The format of the for loop is:

\[
\text{For (startExpression; testExpression; countExpression)} \\
\{ \\
\text{block of code;} \\
\}
\]

The startExpression is evaluated before the loop begins and is almost always an assignment statement (such as \(x = 1;\)). It is possible to declare and assign in the startExpression (such as int \(x = 1;\)). C++ evaluates the startExpression only once, at the beginning of the loop. The countExpression executes after each time the loop repeats, usually incrementing or decrementing the variable. The testExpression evaluates to TRUE (nonzero) or FALSE (zero), then determines whether the body of the loop repeats again. As soon as the testExpression becomes FALSE, C++ stops looping and the program continues with the statement following the forloop.

**While Loop**

The while loop allows programs to repeat a statement or series of statements, over and over, as long as a certain test condition is true. The test condition may be any expression and must be enclosed in parentheses. The block of code is called the body of the loop and is enclosed in braces and indented for readability. Semi-colons follow the statements within the block only. When a program encounters a while loop, the test condition is evaluated first. If the condition is TRUE, the program executes the body of the loop. The program then returns to the test condition and re-evaluates. If the condition is still TRUE, the body executes again. This cycle of testing
and execution continues until the test condition evaluates to 0 or FALSE. 
*The while loop is an entry-condition loop.* If the test condition is FALSE to begin with, the program never executes the body of the loop.

**do-while Loop**

The do-while loop is similar to the while loop, except that the test condition occurs at the end of the loop. Having the test condition at the end guarantees that the body of the loop always executes at least one time.

Syntax:

```
do
{
  block of code;
}
while (test condition);
```

The test condition must be enclosed in parentheses and followed by a semi-colon. Semi-colons also follow each of the statements within the block. The body of the loop (the block of code) is enclosed in braces and indented for readability. *The do-while loop is an exit-condition loop.* This means that the body of the loop is always executed first. Then, the test condition is evaluated. If the test condition is TRUE, the program executes the body of the loop again. If the test condition is FALSE, the loop terminates and program execution continues with the statement following the while.

**Break and continue statement**

Break statement helps to implement premature exit from the loop. It is used with an *if* statement and when the test expression of the *if* statements evaluates to true, the loop exit prematurely.
Example:

```cpp
{
    if (disc < 0)
        break;
    cout << "roots are real";
}
```

**Continue statement**

Continue statement helps to skipping of the statements following it and causes the control to be transferred back to the beginning of the loop.

```cpp
If (disc > 0)
    Continue;
    Cout << "roots are imaginary";
}
```

**Difference between break and continue statement**

<table>
<thead>
<tr>
<th>Break</th>
<th>Continue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Premature exit of loop enclosing it.</td>
<td>Skipping of statements following it in body of loop.</td>
</tr>
<tr>
<td>Can be used with switch statement</td>
<td>Cannot be used with switch statement</td>
</tr>
<tr>
<td>Control is transferred to the statement following the loop</td>
<td>Control is transferred back to the loop</td>
</tr>
<tr>
<td>Loop may not complete the intended number of iterations</td>
<td>Loop completes the intended number of iterations.</td>
</tr>
</tbody>
</table>
INPUT AND OUTPUT STREAM

The default standard output of a program is the screen, and the C++ stream object defined to access it is cout. Cout is used in conjunction with the insertion operator, which is written as << (two “less than” signs).

The default standard input device is usually the keyboard. The standard input in C++ is handled by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream.

ARRAYS

An array is a collection of data storage locations, each of which holds the same type of data. These storage areas are called elements. Arrays are used to handle large amounts of data. An array can be thought of as a collection of numbered boxes each containing one data item. The number associated with the box is the index of the item. To access a particular item the index of the box associated with the item is used to access the appropriate box. The index must be an integer and indicates the position of the element in the array. Thus the elements of an array are ordered by the index. Array subscripts start with the number 0 (not 1). If the name of the array is a, then a [0] is the name of the element that is in position 0, a [1] is the name of the element that is in position 1, etc. In general, the i\textsuperscript{th} element is in position i–1. So if the array has n elements, their names are a [0], a [1], a [2], . . . , a [n – 1].
DECLARATION OF ARRAYS

An array declaration is very similar to a variable declaration. First a type is given for the elements of the array, then an identifier for the array and, within square brackets, the number of elements in the array. The number of elements must be an integer.

The syntax for an array declaration is:

\[
\textit{type array-name [array-size]}
\]

For example data on the average temperature over the year in India for each of the last 100 years could be stored in an array declared as follows:

\[
\textit{float annual_temp[100];}
\]

This declaration will cause the compiler to allocate space for 100 consecutive float variables in memory. The number of elements in an array must be fixed at compile time.

INITIALIZING ARRAYS

An array can be initialized in a similar manner that of variable. In this case the initial values are given as a list enclosed in curly brackets. For example initializing an array to hold the first few prime numbers could be written as follows:

\[
\textit{int prime [ ] = \{1, 2, 3, 5, 7, 11, 13\};}
\]

In this case primes would be allocated space for seven elements. If the array is given a size then this size must be greater than or equal to the number of elements in the initialization list. For example:
\[ \text{int primes[10]} = \{1, 2, 3, 5, 7\}; \]

would reserve space for a ten element array but would only initialize the first five elements.

**Multidimensional Arrays**

Multidimensional arrays allow you to represent grids of data of the same type. A two-dimensional array in C++ is built by using two sets of [ ]’s when you define the array. The first dimension of the array is the row, and the second dimension of the array is the column. The row is always the first dimension in C++. *An array of arrays is called a multidimensional array.* The simplest way to declare a multi-dimensional array as:

\[ \text{int a[30][40][10]}; \]

this is a three-dimensional array with dimensions 30, 40, and 10. The statement \( a[13][10][1] = 100 \) would assign the value 100 to the element identified by the multi-index (13, 10, 1).

**POINTERS**

A *pointer stores addresses in C++*. They are useful for passing parameters into functions in a manner that allows a function to modify and return values to the calling routine. The operating system organizes the memory with unique and consecutive numbers, so if we talk about location 1776 in the memory, we know that there is only one location with that address and also that is between addresses 1775 and 1777. *Pointer variables contain memory addresses as their values.* Whereas a normal variable contains a specific value, a pointer contains the address of a variable that has a
specific value. *When declaring a pointer, an asterisk is placed immediately before the pointer name.*

```c
int *ptr;
```

```c
ptr = &count /* Stores the address of count in ptr */
```

```c
/* The unary operator & returns the address of a variable */
```

To get the value that is stored at the memory location in the pointer it is necessary to dereference the pointer. Dereferencing is done with the unary operator `*`.

**INITIALIZING POINTERS**

Pointers can have any name that is legal for other variables. We can use the convention of naming all pointers with an initial `p`

```c
int age = 30; // declare and initialize a regular int variable
```

```c
int *page = 0; // declare and initialize an integer pointer
```

All pointers, when they are created, should be initialized to some value, even if it is only zero. A pointer whose value is zero is called a null pointer. If the pointer is initialized to zero, you must specifically assign the address to the pointer.

**ARRAYS AND POINTERS**

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, like a pointer is equivalent to the address of the first element that it points to, so
in fact they are the same thing. For example, supposing these two declarations:

\[
\text{int numbers[20];}
\]

\[
\text{int \* p;}
\]

The following allocation would be valid:

\[
\text{p = numbers;}
\]

At this point \( p \) and \( \text{numbers} \) are equivalent and they have the same properties, the only difference is that we could assign another value to the pointer \( p \) whereas \( \text{numbers} \) will always point to the first of the 20 integer numbers of type \( \text{int} \) with which it was defined. So, unlike \( p \), that is an ordinary variable pointer, \( \text{numbers} \) is a constant pointer (indeed an array name is a constant pointer). Therefore, although the previous expression was valid, the following allocation is not valid:

**FUNCTIONS**

A program as consisting of subparts such as obtaining the input data, calculating the output data, displaying the data, etc. In C++, these subparts are called functions. Dividing a program into functions is one of the major principles of top-down, structured programming. Sometimes, it is useful to have a collection of values of different types and to treat the collection as a single item.
PREDEFINED FUNCTIONS

One of the advantages of using functions to divide a programming task into subtasks is that different people can work on different subtasks. C++ comes with libraries of predefined functions that we can use in our programs. For example the Sqrt function calculates the square root of a number. The function requires a number whose square root it has to calculate. This is called argument. Then it computes another value which is the result. This is called value returned. Some functions may have more than one argument but no function can return more than one value. The syntax for using functions is

\[ \text{root} = \text{srt} (25.0); \]

The expression \( \text{srt} (25.0) \) is called a function call. An argument in a function call can be a constant, or a variable or a more complicated expression. We can use function calls wherever we can use an expression.

For example:

\[ \text{earnings} = \text{srt} (\text{sales})/20; \]

Sales and earnings are variable. We can also use a function call directly in a cout statement.

\[ \text{Cout} \ll "\text{The side of a square with area}\" \ll \text{area} \ll "\text{is}\" \ll \text{srt} (\text{area}); \]
THE MAIN () FUNCTION

This is the very first function that we use in any C++ program. This is the starting point of execution of any C++ program. The definition of main () function looks like:

```cpp
main ()
{
    program code
}
```

main () does not return any value but it returns a value of type int to the operating system in C++. So the main () function in C++ is defined as follows:

```cpp
main ()
{
    program code
    return (0);
}
```

FUNCTION DECLARATION, DEFINITION AND CALL

Declaration

We cannot use a function unless we declare it. In function declaration, we give a name to the function, the types of the values returned (if any) and the number and types of the arguments that must be supplied in a call of the function. This is also called function prototyping. When a function is called, the compiler uses the declaration template to ensure that proper arguments are passed, and that the return value is treated properly. Any
violation in matching the arguments or the return types is caught at the
time of compilation by the compiler.
The syntax of a function declaration is

\[ \text{Return-type function-name (argument-list);} \]

These declarations are generally placed before the main part of our
program. Declaring a function as void means that it does not return any
value. A function may be declared more than once in a programme.

**Definition**

Every function that is called must be defined somewhere (only once) in the
program. A function definition is a function declaration in which the body
of the function is presented.

For example:

```c
void change (int*, int*); // declaration
void change (int* p, int* q) // definition
{ int k = *p;
  *p = * q;
  *q = k;
}
```

So a function definition consists of a function header and a body. The only
difference between a function declaration and function header is that the
header does not end with a semicolon. The type of the definition and all
declarations for a function must be the same. The argument names,
however, need not be identical. In the function definition, we require the
variable names because the arguments must be referenced inside the
function. Function definition may be available in a separate file, other than the one having function declaration.

**Function Call**

A function call is an expression consisting of the function name followed by arguments enclosed in parantheses. If there are more than one arguments, they are separated by commas. A function call is an expression that can be used like any other expression. The type of the returned value should match that of the expression. The syntax is:

```
Function-name (argument-list);
```

For example:

```
Side = sqrt (area);
Variable = pow (3.2);
```

**USER DEFINED FUNCTIONS**

We can also define our own functions either in the same file as the main part of the program or in a different file. The definition is the same in either case. The function is called the same way a predefined function is called. The description of a user defined function is given in two parts that are function declaration or prototype and function definition. The declaration and definition of a function are given in the same way. A function definition is like a small program and calling a function is the same thing as running this small program.

**PASSING THE ARGUMENTS**

Whenever a function is called, memory is allocated to its formal parameters and each formal argument is initialized by its corresponding actual
argument. The type of an actual argument is checked against the type of the corresponding formal argument and all standard and user-defined type conversions are performed. There are different rules and ways to pass various arguments.

- **Passing by Value**
  The argument passing that we have discussed till now has been pass-by-value. In this case, the value of the argument is passed to the function. This means that a copy of the argument has been created in the function. Whatever changes we make in the argument in the function does not affect the value of that argument in the calling function. The calling function still retains the value it had before being passed to the function. The disadvantage for this form of passing variables is that this creates copies of the same data in the memory. So if we have a large data, then a lot of memory is wasted to store these argument values. All the operations done on the variable inside the function are done on the copy of the argument and the original value is unchanged.

- **Passing by Reference**
  To avoid the problem of memory wastage and duplication, we use another way of passing the argument. Instead of passing the value of the argument, we pass the address of that argument to the function. In this case, since the address of the argument is passed, the operations performed on the variables inside the function affect the original value of the argument. As a result, the value of the original argument is modified. This is useful in two ways – the duplication and thus wastage of memory is avoided. Also we have mentioned earlier that a function can, under no circumstances, return more than one value. Using this form of passing of arguments, we can accomplish
the task of returning more than one variable value to the calling programme. A pass-by-reference parameter must be marked in some way so that the compiler will be able to differentiate it from the pass-by-value parameter. For this, we indicate a pass-by-reference parameter by attaching the ampersand design & to the end of the type name in the parameter list in both the function

```c
Void f (ints & val, ints ref)
{
    val++;  
    ref++;  
}
```

- **Mixed Parameter List**
  
  We determine whether an argument is passed-by-value or passed-by-reference by finding out whether or not an ampersand is attached to its type specification. We can legitimately mix pass-by-value and pass-by-reference parameters in the same function.

  For example

  ```c
  int g (double & r, float & s, int t)
  {
      r++;  
      s++;  
      t++;  
      return (t);  
  }
  
  In the above example, argument t is passed-by-value and arguments r and s are passed-by-reference.
**INLINE FUNCTIONS**

One of the main objectives why we use functions in our programs is to save memory space. This becomes appreciable when we call a function many times. Whenever we call a function, we waste extra time in jumping to the function, returning the values, etc. One solution to this problem is that we define the function inline. This is most useful when the function definitions are small. In this case, we can give the function definition within the definition of the class. We define an inline function as follows:

```
inline function-header
{
  function-body;
}
```

For example
```
inline int square (int p)
{
  return (p*p);
}
```

The above inline function can be invoked by statements like $x = square (3); y = square (5+2);$ The compiler treats the inline function differently as compared to the normal function. That is why inline functions run more efficiently although they may consume more storage. With an inline function, each function call in your program is replaced by a compiled version of the function definition. So inline function calls do not have the overhead of a normal function call.
All inline functions must be defined before they are called. But we should take care before making a function inline. If the function grows in size, the speed restrictions on how recursive calls are used in function definitions. However, in order for a recursive function definition to be useful, it must be designed so that any call of the function must ultimately terminate with some piece of code that does not depend on recursion. The function may call itself and that recursive call may call the function again. The process may be repeated any number of times. However, the process will finally terminate when one of the recursive calls will finally not depend on recursion in order to return a value.

FRIEND FUNCTIONS

We know that data hiding is an important feature of OOP. It is not possible to access data in the private area of a class by any function which is not a member of that class. Sometimes it is absolutely necessary for a non-member function to access data of private area of a class. To accomplish this task, we define that function as a friend of the class – A friend function of a class is not a member function of that class. But a friend function has access to private data of that class in the same way as a member function has. To make a function friend to a class, you must name it as a friend in the class definition. We do this by listing the function prototype in the definition of the class and placing the keyword friend in front of the function prototype.
The syntax is

```cpp
Class ABC
{
    private:
    - - -
    - - -
    - - -
    public:
    - - -
    - - -
    friend int time (xyz);
}; - - -
```

The function is defined elsewhere in the program. The function definition is the same as any other function definition and does not include the keyword friend. A function can be declared friend to any number of classes. The scope of a friend function is not the class in which it has been declared friend. The friend function is called exactly like an ordinary function and it cannot be called using the object of that class.

**VIRTUAL FUNCTIONS**

C++ allows us to declare a function in the base class that can be redefined in each derived class. The function in the base class, in this case, is declared as virtual by adding the word virtual to the declaration.
RECURSIVE FUNCTIONS

Sometimes, a function definition contains a call to itself. In such cases, the function is called a recursive function. A very commonly used example of this type of function is finding factorial of a number.

FUNCTION OVERLOADING

Most often, it is a good idea to define different names to different function. When some functions conceptually perform the same task on objects of different types, it can be more convenient to give them the same name. If we have two or more function definitions for the same function name, that is called overloading. When we overload a function name, the function definitions must have different numbers of formal parameters or some formal parameters of different types. For example, let us consider the following functions for finding average of two and then three numbers:

```c
Double average (double n1, double n2)
{
  return ((n1+n2)/2.0);
}
double average (double n1, double n2, double n3)
{
  return ((n1+n2+n3)/3.0);
}
```

Here, the function name average has two definitions. This is an example of overloading. As far as the compiler is concerned, the only thing functions of the same name have in common is that name. While overloading a
function name, we have to take care that every function definition has a function prototype. While compilation, the compiler looks at the number of arguments and types of arguments in the function call to decide upon which definition to use. If none of the function prototype matches exactly with the cell, then the compiler gives an error.

Let us consider the following program which uses overloaded function name:

```cpp
#include <iostream.h>
double average (double, double);
double average (double, double, double);
main ( )
{
    cout << "Average of 4.0 and 4.8 is " <<
    average (4.0, 4.8) << endl;
    cout << "Average of 4.0, 4.2 and 4.7 is " <<
    average (4.0, 4.2, 4.7) << endl;
}
double average (double n1, double n2)
{ return ((n1+n2)/2.0);
}
double average (double n1, double n2, double n3)
{return ((n1+n2+n3)/3.0);
}
```

**DEFAULT ARGUMENTS**

C++ allows us to call a function without specifying all its arguments. In such cases, we assign default value to the parameter which may not have a matching argument in the function call. Default values are specified when
the functions are declared. Consider a function for printing an integer. Giving the user an option to choose the base for printing seems reasonable.

\[
\text{Void print (int value, int base = 0); //default base is 10}
\]

\[
\text{Void f ( )}
\]

\[
\text{print (31);}
\]

\[
\text{print (31,10);}
\]

\[
\text{print (31,16);}
\]

\[
\text{print (31,2);}
\]

\[
\text{will give an output as}
\]

\[
31, 31, 1f, 11111
\]

The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point is that only the trailing arguments can have default values. We cannot provide a default value to an argument in the middle of an argument list. Also a default argument cannot be repeated or re-declared in a subsequent declaration in the same scope. Default arguments are useful in situations where some arguments always have the same value. For example interest rates on loans. It provides more flexibility to us. A function can be written with more arguments than required for its most common application.
MODULE-2
**INTRODUCTION TO CLASS AND OBJECTS**

**OBJECTS**

*Object is defined as an identifiable entity with some characteristic and behaviour.* For instance, we can say ‘Orange’ is an object. Its characteristics are: it is spherical shaped, its colour is orange, etc. Its behaviour is: it is juicy and it tastes sweet-sour. While programming using OOP approach, the characteristics of an object are represented by its data and its behaviour is represented by its functions associated. Therefore, in OOP programming object represents an entity that can store data and has its interface through functions. Programming problem is analyzed in terms of objects and the nature of communication between them, so it is necessary for you to understand the nature of communication between objects.

**CLASSES**

*A class is a group of objects that share common properties and relationships* as shown in figure.
A typical class declaration would look like

```cpp
Class item
{
    int number;
    float cost;
    public:
    void getdata(int a, float b);
    void putdata(void);
}
```

For instance, a software model of a car, a car “class”, might contain data about the type of car and abilities such as accelerate or decelerate. A class is a programmer defined data type that has data, its members, and abilities, its methods. **An object is a particular instance of a class.**

A class is a logical method to organize data and functions in the same structure. They are declared using keyword class, whose functionality is similar to that of the C keyword struct, but with the possibility of including functions with a variety of scope as members, instead of only data.

Now let us have a view of a complete class :

```cpp
class class_name{
    permission_label_1:
    member1;
    permission_label_2:
    member2;
    ...
} object_name;
```

where class_name is a name for the class (user defined type) and the optional field object_name is one, or several, valid object identifiers. The body of the declaration can contain members, that can be either data or
function declarations, and optionally permission labels, that can be any of these three keywords: *private, public or protected*. They make reference to the permission which the following members acquire:

- **Private members of a class are accessible only from other members of their same class or from their “friend” classes.**
- **Protected members are accessible from members of their same class and friend classes, and also from members of their derived classes.**
- **Public members are accessible from anywhere the class is visible.**

If we declare members of a class before including any permission label, the members are considered private, since it is the default permission that the members of a class declared with the class keyword acquire.

For example:

```cpp
class crectangle {
    int x, y;
    public:
    void set_values (int,int);
    int area (void);
} rect;
```

Declares class crectangle and an object called rect of this class (type). This class contains four members: two variables of type int (x and y) in the private section (because private is the default permission) and two functions in the public section: set_values() and area(), of which we have only included the prototype. In the previous example, where crectangle was the class name (i.e., the user-defined type), and rect was an object of type crectangle.
PRIVATE AND PROTECTED MEMBERS

The private and protected members of a class can be accessed only by the member function of the class. These members cannot be accessed directly by using the object names. In other words, the names of the private and protected members of the class cannot appear except in the body of the class.

Example:

```cpp
#include <iostream.h>
class x {
private:
  int a;
  void fc (void) {
    cout << a; // private data member (a) used by a member function
  }
public:
  int i;
  void fcc1 (void) {
    cout << 2*i;
    a = 13; // private data member (a) used by a member function
  }
};
x Ob1;
int main (),
{
  Ob1.i= 10;
  Ob1.fcc1();
  Ob1.a = 5; // invalid. a is private data member
  Ob1.fc(); // invalid fc() is a private member function
  .......
}
```
Scope of Public Members

The scope of public members depends upon the object being used for referencing them. If the referencing object is a global object, then the object of public members is also global and if the referencing object is a local object, then the scope of public members is local.

Example

```cpp
#include <iostream.h>
class x
{
private:
int a ;
void fc(void)
{ cout << a;
} 
public :
int i ;
void fcc1( void)
{ cout << 2*i;
 a = 13;
```
fcl();
}
);
x Ob1; // global object Ob1
int main()
{ x Ob2; // local (to main()) object Ob2
 Ob1.i = 10; // Ob1 is global & available to main
 Ob1.fcc1();
 Ob2.i = 20; // Ob2 is local object available to main()
 Ob2.fcc();
}
void func1()
{ x Ob3;
 Ob1.i = 12; // Ob1 is global and hence also available to func1() 
 Ob1.fcc1(); // valid
 Ob2.i = 25; // invalid Ob2 is not available to func1() 
 Ob2.fcc1(); // invalid
 Ob3.i = 15; // valid. Ob3 is locally available to func1() 
 Ob3.fcc1(); // valid
 .............
 .............
}

In the above example, Ob1 is a global object, and so the public members of Ob1 can be accessed from any of the functions in the program. Ob2 is a local object, local to function main(), and thus, the public members of Ob2 can be accessed only in main() and not in func1() and not anywhere else.

GLOBAL CLASS AND LOCAL CLASS

Global Class

A class is said to be a global class if its definition occurs outside the bodies of all functions in a program, which means that the object of this class type can be declared from anywhere in the program.
Example

```cpp
#include <iostream.h>
{
    ......
};
x Ob_1; // global object of type x
int main()
{
x Ob_2; // local object O2 of type x
}
void func1(void)
{
x Ob_3; // local object O3 of type x
    ..........
}
```

In the above program x is a class, defined globally. Thus, an object of type x can be declared from anywhere in the program. The objects Ob_1, Ob_2 and Ob_3 of class type x have been declared at different places in the program.

**Local Class**

A class is said to be a local class if its definition occurs inside a function body, which means that objects of this class type can be declared only within the function that defines this class type.

Example

```cpp
#include <iostream.h>
..................
..................
int main()
{
class y // local class type y
{
    ..................
};
y Ob_1; // local object Ob1 of type y
..................
```
void func1(void)
{ y Ob2; // invalid. Y type is not available in func1()
...............// it is available in main() that has defined it
}

CONSTRUCTORS AND DESTRUCTORS

Constructors

A constructor (having the same name as that of the class) is a member function which is automatically used to initialize the objects of the class type with legal initial values.

These have some special characteristics. These are given below:

1. These are called automatically when the objects are created.
2. All objects of the class having a constructor are initialized before some use.
3. These should be declared in the public section for availability to all the functions.
4. Return type (not even void) cannot be specified for constructors.
5. These cannot be inherited, but a derived class can call the base class constructor.
6. A constructor can call member functions of its class.
7. An object of a class with a constructor cannot be used as a member of a union.
8. A constructor can call member functions of its class.

Destructor

A destructor is called when an object of the class goes out of scope, or when the memory space used by it is deallocated with the help of delete operator.
Some of the characteristics associated with destructors are:

1. *These are called automatically when the objects are destroyed.*
2. *Destructor functions follow the usual access rules as other member functions.*
3. *These de-initialize each object before the object goes out of scope.*
4. *No argument and return type (even void) permitted with destructors.*
5. *These cannot be inherited.*
6. *Static destructors are not allowed.*
7. *Address of a destructor cannot be taken.*
8. *A destructor can call member functions of its class.*
9. *An object of a class having a destructor cannot be a member of a union.*

**OPERATOR OVERLOADING**

In C++ we can multiply two variables of user-defined data type with the same syntax that is applied to the basic data type. This means that C++ has the ability to provide the operators with a special meaning for data type.

*The mechanism which provides this special meaning to operators is called operator overloading.* The operator overloading feature of C++ is one of the methods of realizing polymorphism. Here, poly refers to many or multiple and morphism refers to actions, i.e. performing many actions with a single operator. Thus operator overloading enables us to make the standard operators, like +, -, * etc, to work with the objects of our own data types. So what we do is, write a function which redefines a particular operator so that it performs a specific operation when it is used with the object of a class. Operator overloading is very exciting feature of C++. The concept of operator overloading can also be applied to data conversion. It
enhances the power of extensibility of C++. Thus operator overloading concepts are applied to the following two principle areas:

- Extending capability of operators to operate on user defined data, and
- Data conversion

**Why to Overload Operators?**

The purpose of operator overloading is to make programs clearer by using conventional meanings for ==, [], +, etc. Operators can be overloaded in any way from those available like globally or on the basis of class by class. While implementing the operator overloading this can be achieved by implementing them as functions.

**GENERAL RULES FOR OPERATOR OVERLOADING**

The following rules constrain how overloaded operators are implemented. However, they do not apply to the new and delete operators.

1. You cannot define new operators, such as **.
2. You cannot redefine the meaning of operators when applied to built-in data types.
3. Overloaded operators must either be a non static class member function or a global function.
4. Obey the precedence, grouping, and number of operands dictated by their typical use with built-in types.
5. Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.
6. Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.
7. If an operator can be used as either a unary or a binary operator (&, *, +, and-), you can overload each use separately.
8. Overloaded operators cannot have default arguments.
9. All overloaded operators except assignment (operator=) are inherited by derived classes.
10. Overloading + doesn't overload +=, and similarly for the other extended assignment operators.

<table>
<thead>
<tr>
<th>Table 2.1: List of Operators that can be overloaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
</tr>
<tr>
<td>==</td>
</tr>
<tr>
<td>&lt;=</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>delete[]</td>
</tr>
</tbody>
</table>

Following list shows the operators which can’t be overloaded

```
.  .*  ::  sizeof  ?:
```

**Why Some Operators can’t be Overload?**

Generally the operators that can't be overloaded are like that because overloading them could and probably would cause serious program errors or it is syntactically not possible.

**1. :: (scope resolution), . (member selection), and .* (member selection through pointer to function)**

These operators take a name, rather than a value, as their second operand and provide the primary means of referring to members. C++ has no syntax for writing code that works on names rather than values so syntactically these operators can not be overload.
2. *size of operator*

The size of operator returns the size of the object or type passed as an operand. It is evaluated by the compiler not at runtime so you can not overload it with your own runtime code. It is syntactically not possible to do.

3. *? : (conditional operator)*

All operators that can be overloaded must have at least one argument that is a user-defined type. That means you can't overload that operator which has no arguments.

**DEFINING OPERATOR OVERLOADING**

Overloaded operators are implemented as functions and can be member functions or global functions. An overloaded operator is called an operator function. You declare an operator function with the keyword operator preceding the operator. Overloaded operators are distinct from overloaded functions, but like overloaded functions, they are distinguished by the number and types of operands used with the operator.

**Syntax**

Defining an overloaded operator is like defining a function, but the name of that function is operator #, in which # represents the operator that’s being overloaded.

```cpp
Return_type class_name :: operator op (op_argument_list)
{
    Body of function
}
```
Here return_type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator. Operator op is the name of function. They may be also friend functions. Member function has no argument for unary operator and one argument for binary operator. This is because the object used to invoke the member function is passed implicitly and so it available to member function. This case is not with the friend function. Friend function will have one argument for unary operator and two arguments for binary operator. All the arguments may be passed either by value or by reference.

The steps in overloading the operators are

1. Build a class that defines the data type that is going to use in operation of overloading.
2. Declare the operator function operator op() in public area of class.
3. Now define the operator function to implement the required operations.

**OVERLOADING UNARY OPERATORS**

To declare a unary operator function as a non-static member, you must declare it in the form:

```cpp
return_type operator op();
```

where op is one of the operators listed in the preceding table.

To declare a unary operator function as a global function, you must declare it in the form:

```cpp
return_type operator op( arg );
```
Where op is described for member operator functions and the arg is an argument of class type on which to operate. An overloaded unary operator may return any type.

**INHERITANCE**

Inheritance is a prime feature of object oriented programming language. *It is process by which new classes called derived classes (sub classes, extended classes, or child classes) are created from existing classes called base classes (super classes, or parent classes).* The derived class inherits all the features (capabilities) of the base class and can add new features specific to the newly created derived class. The base class remains unchanged.

Inheritance is a technique of organizing information in a hierarchical form. It is an relation between classes that allows for definition and implementation of one class based on the definition of existing classes.

We can find many real world examples of inheritance like Inheritance between parent and child, employee and manager, person and student, vehicle and light motor vehicle, and animal and mammal etc.

**Derived and Base Class**

As we know, when Class A inherits the feature from class B, then Class A is called the derived class and B is called Base class. A derived class extends its features by inheriting the properties (features) from another class called the base class while adding features of its own.
Declaration of Derived Class Inheritance

The declaration of a derived class shows its relationship with the base class in addition to its own details. The common syntax of declaring a derived class is given as follows:

```cpp
class DerivedClassName : [VisibilityMode] BaseClassName
{
    // members of derived class
};
```

The derivation of DerivedClassName from the BaseClassName is indicated by colon (:). The VisibilityMode enclosed within the square bracket is optional. If the VisibilityMode is specified, it must be either public or private or protected. It specifies the features of the base class that are privately derived or publicly derived.

Visibility of Class Members

There are three visibility modes (visibility modifier). They are private, public and protected.

*Why is the different type of visibility mode needed in derivation of a derived class?*

A class may contain some secret information which we are not interested to share by the derived classes and non-secret information which we are interested to share by the derived class. Visibility mode promotes encapsulation. The visibility of the base class members undergoes modification in a derived class as summarized in Table.
In derived class declaration, if the visibility mode is private then both „public members“ of the base class as well as protected members” of the base class will become private members of the derived class. Therefore, both public and protected member of base class can only be accessed by the member functions of the derived class. They can not be accessed by the objects of the derived class. And private members of the base class will not be inherited. On the other hand, if visibility mode is public, public members of the base class will become public members of the derived class and protected members of the base class will become protected members of the derived class whereas private member of the base class will never become the members of the declared class i.e. it will not be inherited. If the visibility mode is protected then the public and protected members of the base class will become the protected members of the derived class. In this case also, the private members of the base class will not become the member of its derived class.

### TYPES OF INHERITANCE

1. **Single Inheritance:**

   Derivation of a class from only one base class is called a single inheritance.

2. **Multiple Inheritance:**
Derivation of a class from several (two or more) base classes is called multiple inheritance.

3. **Multi-level Inheritance:**

Derivation of a class from another derived class is called multilevel inheritance.

4. **Hierarchical Inheritance:**

Derivation of several classes from a single base class is called hierarchical inheritance.

5. **Hybrid Inheritance:**

Derivation of a class involving more than one form of inheritance is called hybrid inheritance.

6. **Multi-path Inheritance:**

Derivation of a class from other derived classes, which are derived from the same base class is called multi-path inheritance.

**SINGLE INHERITANCE**

In Single Inheritance, derived class inherits the feature of one base class. If a class is derived from one base class, it is called Single Inheritance.
Class A is the base class and class B is the derived class. The following are the common steps to implement an inheritance. First, declare a base class and second declare a derived class. The syntax of single inheritance

```cpp
class A
{
    // members of class A
};
class B : [public/private/protected] A
{
    // members of class B
};
```

**MULTIPLE INHERITANCE**

In multiple inheritance, derived class inherits features from more than one parent classes (base classes). In other way if a class is derived from more than one parent class (base classes), then it is called multiple inheritance
The syntax of declaration of Multiple Inheritance is given below:

```c
class A
{
    // members of class A
};

class B
{
    // members of class B
};

{
    // members of class C
};
```

**MULTI-LEVEL INHERITANCE**

In multi-level inheritance, the class inherits the feature of another derived class. If a class C is derived from class B which in turn is derived from class A and so on. It is called multi-level inheritance.
A stream is a logical device that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same way even though the actual physical devices they are connected to may differ substantially. Because all streams behave the same, the same I/O functions can operate on virtually any type of physical device. For example, one can use the same function that writes to a file to write to the printer or to the screen. The advantage to this approach is that you need learn only one I/O system.

A stream acts like a source or destination. The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called the output stream. C++ contains cin and cout predefined streams that opens automatically when a program begins its execution. cin represents the input stream connected to the standard input device and cout represents the output stream connected to standard output device.

**C++ STREAM CLASSES**

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes. Figure shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file iostream. The file should be included in all programs that communicate with the console unit.
As in figure ios is the base class for istream(input stream) and ostream(output stream) which are base classes for iostream(input/output stream). The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream. The class ios provides the basic support for formatted and unformatted input/output operations. The class istream provides the facilities for formatted and unformatted input while the class ostream(through inheritance) provides the facilities for formatted output. The class iostream provides the facilities for handling both input output streams.

<table>
<thead>
<tr>
<th>Class name</th>
<th>Contents</th>
</tr>
</thead>
</table>
| ios (General input/output stream class) | Contains basic facilities that are used by all other input and output classes  
  Also contains a pointer to buffer object (streambuf object)  
  Declares constants and functions that are necessary for handling formatted input and output operations |
| istream(input stream)    | Inherits the properties of ios  
  Declares input functions such as get(), getline(), and read()  
  Contains overloaded extraction operator (>) |
| ostream(output stream)   | Inherits the property of ios  
  Declares output functions put() and write()  
  Contains overloaded insertion operator (<) |
| iostream (input/output stream) | Inherits the properties of ios stream and ostream through multiple inheritance and thus contains all the input and output functions |
| streambuf               | Provides an interface to physical devices through buffer  
  Acts as a base for filebuf class used for files |
MODULE-3
NUMERICAL METHODS

Numerical methods are methods for solving problems on computers by numerical calculations, often giving a table of numbers and/or graphical representations or figures. Numerical methods tend to emphasize the implementation of algorithms. The aim of numerical methods is therefore to provide systematic methods for solving problems in a numerical form. The process of solving problems generally involves starting from an initial data, using high precision digital computers, following the steps in the algorithms, and finally obtaining the results.

NUMBER REPRESENTATION

A base-b number is made up of individual digits. For decimal numbers, the base (radix) is 10. There are two binary digits (bits) in the binary number system: zero and one. The left most bit is called the most significant bit (MSB) and the right most bit is the least significant bit (LSB).

Most computer languages use FLOATING-POINT ARITHMETIC. Every number is represented using a (fixed, finite) number of binary digits, called bits. Each binary digit is referred to as a bit. In this method, the computer represents a number in the following form:

\[ \text{Number} = \sigma \, mb^{t-p} \]

where \( \sigma \) = sign of the number (±), denoted by a single bit.

\( m \) = mantissa or a fraction (a value which lies between 0.1 and 1).
\[ b = \text{the base of the internal number system (} b = 2 \text{ for binary, } b = 10 \text{ for decimal or } b = 16 \text{ for hexadecimal computers).} \]

\[ t = \text{shifted exponent (the value that is actually stored).} \]

\[ p = \text{shift required to recover the actual exponent. Shifting in the exponent is normally done to avoid the need for a sign bit in the exponent itself.} \]

The number is then stored by storing only the values of \( \sigma \), \( m \) and \( t \). The normal way to represent and store numbers is to use a binary or base 2 number system which contains the following two digits.

**Binary digits = \{0 1\}**

Conversion between base 10 and base 2 is performed automatically by programming languages.

Thus, conversion of an \( n \)-bit binary integer \( b = b_{n-1} \ldots b_0 \) to its decimal equivalent \( x \) is done as a sum of \( n \) powers of 2:

\[
x = \sum_{k=0}^{n-1} b_k 2^k \]

For example to determine the decimal values of \( x = (10010110)_2 \)

\[
x = \sum_{k=0}^{7} b_k 2^k \text{ using equation}
\]

\[ = 2^1 + 2^2 + 2^4 + 2^7 \]

\[ = 2 + 4 + 16 + 128 = 150 \]
**Table 1.1: Binary, Decimal, and Hexadecimal Numbers**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00</td>
<td>0</td>
<td>1000</td>
<td>08</td>
<td>8</td>
</tr>
<tr>
<td>0001</td>
<td>01</td>
<td>1</td>
<td>1001</td>
<td>09</td>
<td>9</td>
</tr>
<tr>
<td>0010</td>
<td>02</td>
<td>2</td>
<td>1010</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>0011</td>
<td>03</td>
<td>3</td>
<td>1011</td>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>0100</td>
<td>04</td>
<td>4</td>
<td>1100</td>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>0101</td>
<td>05</td>
<td>5</td>
<td>1101</td>
<td>13</td>
<td>D</td>
</tr>
<tr>
<td>0110</td>
<td>06</td>
<td>6</td>
<td>1110</td>
<td>14</td>
<td>E</td>
</tr>
<tr>
<td>0111</td>
<td>07</td>
<td>7</td>
<td>1111</td>
<td>15</td>
<td>F</td>
</tr>
</tbody>
</table>

Ex: To Convert \((1011)_2\) to base-10

\[(2)^3 + (0)(2)^2 + (1)(2)^1 + 1 = 11\]

**SIGNIFICANT DIGITS**

The digits that are used to express a number are called significant digits. Figure is synonymous with digit. A significant digit of an approximate number is any non-zero digit in its decimal representation, or any zero lying between significant digits, or used as place holder to indicate a retained place.

**ERRORS**

**Sources of Errors**

When a computational procedure is involved in solving a scientific-mathematical problem, errors often will be involved in the process. A rough classification of the kinds of original errors that might occur is as follows:


- **Modelling Errors**: Mathematical modelling is a process when mathematical equations are used to represent a physical system. This modelling introduces errors and are called modelling errors.

- **Blunders and Mistakes**: Blunders occur at any stage of the mathematical modelling process and consist to all other components of error. Blunders can be avoided by sound knowledge of fundamental principles and with taking proper care in approach and design to a solution. Mistakes are due to the programming errors.

- **Machine Representation and Arithmetic Errors**: These errors are inevitable when using floating-point arithmetic when using computers or calculators. Examples are rounding and chopping errors.

- **Mathematical Approximation Errors**: This error is also known as a truncation error or discretisation error. These errors arise when an approximate formulation is made to a problem that otherwise cannot be solved exactly.

Errors are introduced by the computational process itself. Computers perform mathematical operations with only a finite number of digits. If the number $x_a$ is an approximation to the exact result $x_e$.

Then the difference $x_e - x_a$ is called error. Hence,

$$\text{Exact value} = \text{approximate value} + \text{error}$$

**TYPES OF ERRORS**

In numerical computations, we come across the following types of errors:

(a) Absolute and Relative errors
(b) Inherent errors
(c) Round-off errors
(d) Truncation errors

**Absolute and Relative Errors**

If \( X_E \) is the exact or true value of a quantity and \( X_A \) is its approximate value, then \( |X_E - X_A| \) is called the absolute error \( E_a \). Therefore absolute error

\[
E_a = |X_E - X_A|
\]

The relative error is defined by

\[
E_r = \left| \frac{X_E - X_A}{X_E} \right|
\]

The percentage relative error is

\[
E_p = 100E_r = 100 \left| \frac{X_E - X_A}{X_E} \right|
\]

**Inherent Errors**

Inherent errors are the errors that pre-exist in the problem statement itself before its solution is obtained. Inherent errors exist because the data being approximate or due to the limitations of the calculations using digital computers. Inherent errors cannot be completely eliminated but can be minimised if we select better data or by employing high precision computer computations.
**Round-off Errors**

Round-off error is due to the inaccuracies that arise due to a finite number of digits of precision used to represent numbers. All computers represent numbers, except for integer and some fractions, with imprecision. Digital computers use floating-point numbers of fixed word length. This type of representation will not express the exact or true values correctly. Error introduced by the omission of significant figures due to computer imperfection is called the round-off error.

Round-off errors are avoidable in most of the computations. When n digits are used to represent a real number, then one method is keep the first n digits and chop off all remaining digits. Another method is to round to the nth digit by examining the values of the remaining digits. Errors which result from this process of chopping or rounding method are known as round-off errors.

**Truncation Errors**

Truncation errors are defined as those errors that result from using an approximation in place of an exact mathematical procedure. Truncation error results from terminating after a finite number of terms known as formula truncation error or simply truncation error.
ERROR PROPAGATION

Our major concern is how an error at one point in the process propagates and how it affects the final total error.

For addition and subtraction the magnitude of the absolute error of a sum or difference is equal to or less than the sum of the magnitudes of the absolute errors of operands. This inequality is called triangle inequality. The equality applies when the operands having the same sign and the inequality applies if the signs are different.

ERROR ESTIMATION

Few computer methods are available to provide error estimates.

1. Double precision method

In this method, the problem is solved twice, once in single precision and then in double precision. The estimate on the round-off error is then simply given by the difference between the two results obtained.

2. Interval arithmetic method

Each number in this method is represented by two machine numbers corresponding to the estimated maximum and minimum values. Two solutions are obtained at every step corresponding to the maximum and minimum values. The true solution is assumed to lie in about the centre of the range. The range here is the difference between the solutions corresponding to the maximum and minimum values.
3. Significant digit arithmetic method

In this method, the digits lost due to the subtraction of two nearly equal machine numbers are tracked. Only the significant digits in a number are kept and the rest are rejected or ignored. In this way, all digits retained or kept are assumed to be significant. The results obtained with this method are considered to be very conservative.

4. Statistical approach

This method starts with the assumption that the round-off error is independent. A stochastic model for the propagation of round-off errors is then adapted in which the local errors are considered as random variables. The local round-off errors are assumed to be either uniformly or normally distributed between their extreme values. Using standard statistical analysis methods, the standard deviation, the variance and the accumulated round-off error are estimated.

5. Backward error analysis

In this method, based on the result of a computation the possible range of input data that could have produced it is determined. If the results found with this approach is consistent with the input data, within the range of observational or round-off error, then there is some confidence is placed on the result. If this does not happen, then a major source of error is assumed to exist somewhere else, presumably within the algorithm itself.

6. Forward error analysis:

The method can be illustrated by means of an example.
Suppose the value of $A(B + C)$ is to be computed when $a$, $b$ and $c$ are the approximations to $A$, $B$ and $C$ respectively, and the respective error amounts are $e_1$, $e_2$ and $e_3$.

The true value is

$$A(B + C) = (a + e_1)(b + e_2 + c + e_3) = ab + ac + \text{error}$$

where $\text{error} = a(e_2 + e_3) + be_1 + ce_1 + e_1e_2 + e_1e_3$

Now assuming the uniform bound $|e_i| \leq \varepsilon$ and that error products can be ignored, we get

$$|\text{error}| \leq [2|a| + |b| + |c|] \varepsilon$$

This procedure can be carried out for any algorithm. It is a tedious analysis. The resulting bounds are generally very conservative.

### CONDITIONING AND STABILITY

A computation process in which the cumulative effect of all input errors is grossly magnified is said to be *numerically unstable*. The investigations to see how the small changes in input parameters influence the output are termed as sensitivity analysis. Numerical instability may arise due to sensitivity inherent in the problem or sensitivity of the numerical method.

A mathematical model can be solved either by analytical methods or by numerical methods. In either case when a small disturbance in an input parameter causes unacceptable amount of error in the output we can say that the problem is *inherently unstable*. Such problems are said to be *ill conditioned*.

The term *condition* is used to describe the sensitivity of problems or methods to uncertainty. We can quantify the condition by using *condition number*. 
A condition number is often defined as the ratio of the relative errors.

\[
\text{Condition number} = \frac{\frac{a f'(a)}{f(a)}}
\]

The condition number indicates the extent to which an uncertainty in \(x\) is magnified by \(f(x)\).

- \(\text{Condition number} = 1\) (function’s relative error = relative error in \(x\))
- \(\text{Condition number} > 1\) (relative error is amplified)
- \(\text{Condition number} < 1\) (relative error is attenuated)
- \(\text{Condition number} > \text{very large number}\) (the function is ill-conditioned)

### CONVERGENCE

Most of the numerical computing processes acquired re iterative in nature. The number of iterations required to reach the given limit depends on the rate at which the iterates converge to the result. The higher the order of iteration more rapid is the rate of convergence.

The rate of convergence is a measure of how fast the truncation error goes to zero. This measure is used to comparing various iterative methods.

### CONTROL OF ERRORS

The total error can be controlled or minimised by reducing truncation and round off errors. It can be achieved by

- Increasing the significant figures of the computer.
- Minimising the number of arithmetic operations.
• Avoiding subtractive cancellations.
• Choosing proper initial parameters.
• Reasonable understanding of the problem.
• Refining and enlarging the mathematical models by incorporating more features.
• Use of good programming techniques.

INTERPOLATION

Interpolation is the technique of estimating the value of a function for any intermediate value of the independent variable. The process of computing or finding the value of a function for any value of the independent variable outside the given range is called extrapolation. Here, interpolation denotes the method of computing the value of the function \( y = f(x) \) for any given value of the independent variable \( x \) when a set of values of \( y = f(x) \) for certain values of \( x \) are known or given.

If the function \( f(x) \) is known explicitly, then the value of \( y \) corresponding to any value of \( x \) can easily be obtained. On the other hand, if the function \( f(x) \) is not known, then it is very hard to find the exact form of \( f(x) \) with the tabulated values \( (x_i, y_i) \). In such cases, the function \( f(x) \) can be replaced by a simpler, function, say, \( \varphi(x) \), which has the same values as \( f(x) \) for \( x_0, x_1, x_2, \ldots, x_n \). The function \( \varphi(x) \) is called the interpolating or smoothing function.

If \( \varphi(x) \) is a polynomial, then \( \varphi(x) \) is called the interpolating polynomial and the process of computing the intermediate values of \( y = f(x) \) is called the polynomial interpolation.
LANGRANGE INTERPOLATION

Let the data

<table>
<thead>
<tr>
<th>$x$</th>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>...</th>
<th>$x_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$</td>
<td>$f(x_0)$</td>
<td>$f(x_1)$</td>
<td>$f(x_2)$</td>
<td>...</td>
<td>$f(x_n)$</td>
</tr>
</tbody>
</table>

be given at distinct unequally spaced points or non-uniform points $x_0, x_1, ..., x_n$. This data may also be given at evenly spaced points.

For this data, we can fit a unique polynomial of degree $\leq n$. Since the interpolating polynomial must use all the ordinates $f(x_0), f(x_1), ..., f(x_n)$, it can be written as a linear combination of these ordinates. That is, we can write the polynomial as

$$P_n(x) = l'_0(x) f(x_0) + l'_1(x) f(x_1) + ... + l'_n(x) f(x_n)$$

$$= l'_0(x) f(x_0) + l'_1(x) f(x_1) + ... + l'_n(x) f(x_n)$$

where $f(x'_i) = f_i$ and $l'_i(x), i = 0, 1, 2, ..., n$ are polynomials of degree $n$. This polynomial fits the data given in (2.1) exactly.

At $x = x_0$, we get

$$f(x_0) = P_n(x_0) = l'_0(x_0) f(x_0) + l'_1(x_0) f(x_1) + ... + l'_n(x_0) f(x_n).$$

This equation is satisfied only when $l'_0(x_0) = 1$ and $l'_i(x_0) = 0, i \neq 0$.

At a general point $x = x_p$, we get

$$f(x_p) = P_n(x_p) = l'_0(x_p) f(x_0) + ... + l'_i(x_p) f(x_i) + ... + l'_n(x_p) f(x_n).$$

This equation is satisfied only when $l'_i(x_i) = 1$ and $l'_j(x_j) = 0, i \neq j$.

Therefore, $l'_i(x)$, which are polynomials of degree $n$, satisfy the conditions

$$l'_i(x_j) = \begin{cases} 0, & i \neq j \\ 1, & i = j. \end{cases}$$

Since, $l'_i(x) = 0$ at $x = x_0, x_1, ..., x_{i-1}, x_{i+1}, ..., x_n$, we know that

$$(x - x_0), (x - x_1), ..., (x - x_{i-1}), (x - x_{i+1}), ..., (x - x_n)$$

are factors of $l'_i(x)$. The product of these factors is a polynomial of degree $n$. Therefore, we can write

$$l'_i(x) = C(x - x_0)(x - x_1)...(x - x_{i-1})(x - x_{i+1})...(x - x_n)$$
where $C$ is a constant.

Now, since $l_i(x_i) = 1$, we get

$$l_i(x_i) = 1 = C(x_i - x_0)(x_i - x_1)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n).$$

Hence,

$$C = \frac{1}{(x_i - x_0)(x_i - x_1)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n)}.$$

Therefore,

$$l_i(x) = \frac{(x - x_0)(x - x_1)...(x - x_{i-1})(x - x_{i+1})...(x - x_n)}{(x_i - x_0)(x_i - x_1)...(x_i - x_{i-1})(x_i - x_{i+1})...(x_i - x_n)}.$$

**Linear interpolation**

For $n = 1$, we have the data

<table>
<thead>
<tr>
<th>$x$</th>
<th>$x_0$</th>
<th>$x_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$</td>
<td>$f(x_0)$</td>
<td>$f(x_1)$</td>
</tr>
</tbody>
</table>

The Lagrange fundamental polynomials are given by

$$l_0(x) = \frac{(x - x_1)}{(x_0 - x_1)}, l_1(x) = \frac{(x - x_0)}{(x_1 - x_0)}.$$

The Lagrange linear interpolation polynomial is given by

$$P_1(x) = l_0(x) f(x_0) + l_1(x) f(x_1).$$

**Quadratic interpolation**

For $n = 2$, we have the data

<table>
<thead>
<tr>
<th>$x$</th>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$</td>
<td>$f(x_0)$</td>
<td>$f(x_1)$</td>
<td>$f(x_2)$</td>
</tr>
</tbody>
</table>

The Lagrange fundamental polynomials are given by

$$l_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}, l_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}, l_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}.$$

The Lagrange quadratic interpolation polynomial is given by

$$P_2(x) = l_0(x) f(x_0) + l_1(x) f(x_1) + l_2(x) f(x_2).$$
NEWTON’S DIVIDED DIFFERENCE INTERPOLATION

Divided differences

Let the data, \((x_i, f(x_i)), i = 0, 1, 2, ..., n,\) be given. We define the divided differences as follows.

First divided difference Consider any two consecutive data values \((x_i, f(x_i)), (x_{i+1}, f(x_{i+1})).\) Then, we define the first divided difference as

\[
f[x_i, x_{i+1}] = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}, \quad i = 0, 1, 2, ..., n - 1.
\]

Therefore,

\[
f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}, \quad f[x_1, x_2] = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \text{ etc.}
\]

Note that

\[
f[x_i, x_{i+1}] = f[x_{i+1}, x_i] = \frac{f(x_i)}{x_i - x_{i+1}} + \frac{f(x_{i+1})}{x_{i+1} - x_i}.
\]

The \(n\)th divided difference using all the data values in the table, is defined as

\[
f[x_0, x_1, ..., x_n] = \frac{f[x_1, x_2, ..., x_n] - f[x_0, x_1, ..., x_{n-1}]}{x_n - x_0}
\]

<table>
<thead>
<tr>
<th>(x)</th>
<th>(f(x))</th>
<th>First d.d</th>
<th>Second d.d</th>
<th>Third d.d</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_0)</td>
<td>(f_0)</td>
<td>(f[x_0, x_1])</td>
<td>(f[x_0, x_1, x_2])</td>
<td>(f[x_0, x_1, x_2, x_3])</td>
</tr>
<tr>
<td>(x_1)</td>
<td>(f_1)</td>
<td>(f[x_1, x_2])</td>
<td>(f[x_1, x_2, x_3])</td>
<td>(f[x_1, x_2, x_3])</td>
</tr>
<tr>
<td>(x_2)</td>
<td>(f_2)</td>
<td>(f[x_2, x_3])</td>
<td>(f[x_2, x_3])</td>
<td>(f[x_2, x_3])</td>
</tr>
<tr>
<td>(x_3)</td>
<td>(f_3)</td>
<td>(f[x_3, x_4])</td>
<td>(f[x_3, x_4])</td>
<td>(f[x_3, x_4])</td>
</tr>
</tbody>
</table>

NEWTON’S DIVIDED DIFFERENCE INTERPOLATION

We mentioned earlier that the interpolating polynomial representing a given data values is unique, but the polynomial can be represented in various forms.

We write the interpolating polynomial as

\[
f(x) = P_n(x) = c_0 + (x - x_0)c_1 + (x - x_0)(x - x_1)c_2 + \ldots + (x - x_0)(x - x_1)\ldots(x - x_{n-1})c_n.
\]
The polynomial fits the data \( P_n(x_i) = f(x_i) = f_i \).

Setting \( P_n(x_0) = f_0 \), we obtain

\[
P_n(x_0) = f_0 = c_0
\]

since all the remaining terms vanish.

Setting \( P_n(x_1) = f_1 \), we obtain

\[
f_1 = c_0 + (x_1 - x_0) c_1, \quad \text{or} \quad c_1 = \frac{f_1 - c_0}{x_1 - x_0} = \frac{f_1 - f_0}{x_1 - x_0} = f [x_0, x_1].
\]

Setting \( P_n(x_2) = f_2 \), we obtain

\[
f_2 = c_0 + (x_2 - x_0) c_1 + (x_2 - x_1)(x_2 - x_1) c_2,
\]

or

\[
c_2 = \frac{f_2 - f_0 - (x_2 - x_0) f [x_0, x_1]}{(x_2 - x_0)(x_2 - x_1)} = \frac{1}{(x_2 - x_0)(x_2 - x_1)} \left[ f_2 - f_0 - (x_2 - x_0) \left( \frac{f_1 - f_0}{x_1 - x_0} \right) \right]
\]

\[
= \frac{f_0}{(x_0 - x_1)(x_0 - x_2)} + \frac{f_1}{(x_1 - x_0)(x_1 - x_2)} + \frac{f_2}{(x_2 - x_0)(x_2 - x_1)} = f [x_0, x_1, x_2].
\]

By induction, we can prove that

\[
c_n = f [x_0, x_1, x_2, ..., x_n].
\]

Hence, we can write the interpolating polynomial as

\[
f(x) = P_n(x)
\]

\[
= f(x_0) + (x - x_0) f [x_0, x_1] + (x - x_0)(x - x_1) f [x_0, x_1, x_2] + ... + (x - x_0)(x - x_1)...(x - x_{n-1}) f [x_0, x_1, ..., x_n]
\]

**This polynomial is called the Newton’s divided difference interpolating polynomial.**
Inverse interpolation

Suppose that a data \((x_i, y_i), i = 0, 1, 2, ..., n\), is given. In interpolation, we predict the value of the ordinate \(f(x')\) at a non-tabular point \(x = x'\). In many applications, we require the value of the abscissa \(x'\) for a given value of the ordinate \(f(x')\). For this problem, we consider the given data as \((f(x_i), x_i), i = 0, 1, 2, ..., n\) and construct the interpolation polynomial. That is, we consider \(f(x)\) as the independent variable and \(x\) as the dependent variable. This procedure is called inverse interpolation.

Langrange Formula for inverse interpolation

In Langrange interpolation formula \(y\) is expressed as a function of \(x\) as

\[
y = f(x) = \frac{(x-x_1)(x-x_2)...(x-x_n)}{(x_0-x_1)(x_0-x_2)...(x_0-x_n)} y_0 + \frac{(x-x_0)(x-x_2)...(x-x_n)}{(x_1-x_0)(x_1-x_2)...(x_1-x_n)} y_1 + ...
\]

\[
+ \frac{(x-x_0)(x-x_1)...(x-x_{n-1})}{(x_n-x_0)(x_n-x_1)...(x_n-x_{n-1})} y_n
\]

By interchanging \(x\) and \(y\) in Eq.(5.46) we can express \(x\) as a function of \(y\) as follows:

\[
x = \frac{(y-y_1)(y-y_2)...(y-y_n)}{(y_0-y_1)(y_0-y_2)...(y_0-y_n)} x_0 + \frac{(y-y_0)(y-y_2)...(y-y_n)}{(y_1-y_0)(y_1-y_2)...(y_1-y_n)} x_1 + ...
\]

\[
+ \frac{(y-y_0)(y-y_1)...(y-y_{n-1})}{(y_n-y_0)(y_n-y_1)...(y_n-y_{n-1})} x_n
\]

INTERPOLATION WITH EQUAL INTERVALS

Newton’s Forward Interpolation Formula

Let \(y = f(x)\), which takes the values \(y_0, y_1, y_2, ..., y_n\), that is the set of \((n + 1)\) functional values \(y_0, y_1, y_2, ..., y_n\), \(y_0\) are given corresponding to the set of \((n + 1)\) equally spaced values of the independent variable, \(x_i = x_0 + ih, i = 0, 1, 2, ..., n\) where \(h\) is the spacing. Let \(\phi(x)\) be a polynomial of the \(n^{th}\) degree in \(x\) taking the same values as \(y\) corresponding to \(x = x_0, x_1, ..., x_n\). Then, \(\phi(x)\) represents the continuous function \(y = f(x)\) such that \(f(x_i) = \phi(x_i)\) for \(i = 0, 1, 2, ..., n\) and at all other points \(f(x) = \phi(x) + R(x)\) where \(R(x)\) is called the error term (remainder term) of the interpolation formula.

Let

\[
\phi(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)(x - x_1)(x - x_2) + \cdots + a_n(x - x_0)(x - x_1)(x - x_2) \cdots (x - x_{n-1})
\]

and

\[
\phi(x_i) = y_i; \ i = 0, 1, 2, ..., n
\]

The constants \(a_0, a_1, a_2, ..., a_n\) can be determined as follows:

Substituting \(x = x_0, x_1, x_2, ..., x_n\) successively in Eq.(5.24), we get
The formula is called the Newton’s forward interpolation formula. This formula is used to interpolate the values of \( y \) near the beginning of a set of
equally spaced tabular values. This formula can also be used for extrapolating the values of \( y_a \) little backward of \( y_0 \).

**Newton’s Backward Interpolation Formula**

Newton’s forward interpolation formula is not suitable for interpolation values of \( y \) near the end of a table of values.

\[
\phi(x) = y_n + \frac{\nu(v+1)}{2!} \nabla^2 y_n + \ldots + \nu(v+1) \ldots \frac{(\nu+n-1)}{n!} \nabla^n y_n
\]

where \( \nu = \frac{x - x_n}{h} \)

The formula given is called the Newton’s backward interpolation formula. This formula is used for interpolating values of \( y \) near the end of the tabulated values and also used for extrapolating values of \( y_a \) little backward of \( y_n \).

**HERMITE’S INTERPOLATION FORMULA**

Hermite’s interpolation formula provides an expression for a polynomial passing through given points with given slopes. The Hermite interpolation accounts for the derivatives of a given function. Let \( x_n, f_i, f_i' \) (for \( i = 0, 1, 2, ..., n \)) be given.

The polynomial \( f(x) \) of degree \( (2n + 1) \) for which \( f(x_i) = f_i \) and \( f'(x_i) = f_i' \) is given by:

\[
f(x) = \sum_{j=0}^{n} h_j(x) f_j + \sum_{j=0}^{n} \bar{h}_j(x) f_j'
\]

where

\[
h_j(x) = 1 - \frac{q_n(x_j)}{q_n(x_j)}(x - x_j)[L_j(x)]^2
\]

\[
\bar{h}_j(x) = (x - x_j)[L_j(x)]^2
\]

\[
q_n(x) = (x - x_0)(x - x_1)\ldots(x - x_n)
\]

\[
L_j(x) = \frac{q_n(x)}{(x - x_j)q_n(x_j)}
\]

**PROGRAM FOR LANGRANGE INTERPOLATION**
To interpolate the value of $f(x)$ at any point $x$ using Newton-Gregory forward interpolation formula, when $n$ tabulated values of $x$ and $f(x)$ are given at regular intervals.

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

void main()
{
    int n, i, j;
    float x[10], f[10], d[10][10];
    float xx, fx, prod, t, h;
    puts("\nHow many tabulated values? <= 10");
    scanf("%d", &n);
    puts("Give the tabulated values of x and f(x): ");
    for (i=0; i <= (n-1); i++)
        scanf("%f %f", &x[i], &f[i]);
    h = x[1] - x[0];
    puts("Give value of x where f(x) is to be computed:" );
    scanf("%f", &xx);
    t = (xx - x[0]) / h;
    for (j=0; j <= (n-1); j++)
        for (i=0; i <= (n-1); i++)
            d[i][j] = 0;
    j=0;
    for (i=0; i <= (n-1); i++)
        d[j][i] = f[i];
    for (j=1; j <= (n-1); j++)
    {
        for (i=0; i <= (n-j-1); i++)
            d[j][i] = d[j-1][i+1] - d[j-1][i];
    }
    fx = f[0];
    for (i = 1; i <= (n-1); i++)
    {
        prod = d[i][0];
        for (j=1; j <= i; j++)
            prod *= (t-j+1)/j;
        fx += prod;
    }
}``
printf ("\nValue of f(x) = %f corresponding to x = %f", fx
xx);

getch();
}

Sample Output

How many tabulated values? <= 10

6

Give the tabulated values of x and f(x)

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>184</td>
</tr>
<tr>
<td>50</td>
<td>204</td>
</tr>
<tr>
<td>60</td>
<td>226</td>
</tr>
<tr>
<td>70</td>
<td>250</td>
</tr>
<tr>
<td>80</td>
<td>276</td>
</tr>
<tr>
<td>90</td>
<td>304</td>
</tr>
</tbody>
</table>

Give value of x where f(x) is to be computed:

43

Value of f(x) = 189.789993 corresponding to x = 43.000000

**AIKEN INTERPOLATION FORMULA**

This is an iterative interpolation method, using a recursive polynomial construction. The Aitken’s method is efficient as the Newton’s formula, but is more easier to implement in a computer program. It is an example of how iteration can be used to construct a sequence of polynomials of increasing order.

Here, a number of interpolation points are initially considered and if they are not sufficient, new points can be added iteratively until the desired accuracy is achieved. The addition of new interpolation points cannot be done directly in Lagrangian interpolation. The Aitken's method is much similar to the Neville interpolation method.

Let \((x_0, y_{0,0}), (x_1, y_{1,0}), (x_2, y_{2,0}), \ldots, (x_n, y_{n,0})\) be the \(n\) given data points and it is needed to interpolate the value of \(y_p = f(x_p)\) at the point, \(x_p\). The iterative formula for the method is:

\[
y_p = y_{i,j} = \frac{y_{j-1,j-1} \times (x_i - x_p) - y_{i,j-1} \times (x_{j-1} - x_p)}{x_i - x_{j-1}}
\]

where

\[
y_{i,j} = \frac{1}{x_i - x_{j-1}} \begin{vmatrix} y_{j-1,j-1} & x_{j-1} - x_p \\ y_{i,j-1} & x_i - x_p \end{vmatrix} \quad \text{for} \quad j = 1, 2, \ldots, n-1
\]

\[
y_{i,0} = f(x_i) \quad \text{for} \quad i = 0, 1, \ldots, n
\]

The formula is valid for \(i = j\).

\[(5.12)\]
Using the given data, the Aitken’s table is developed with the help of the above formula. The first two columns in the table represent the given data. Each new element in the subsequent columns is computed using the elements in the \{same row, preceding column\} and \{top row, preceding column\}. Values in the third column are computed using two sets of given data each, while three sets of data are utilised for computing values in the fourth column. Each iteration results in more refined values of $y_f$. In the table, $y_{n-1,n}$, is computed using the unique polynomial of $n^{th}$ order coinciding with $f(x)$ at the points $x_0, x_1, x_2, \ldots, x_{n-1}$.

Table 5.5 Aitken’s table

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$y_{p,i=1}$</th>
<th>$y_{p,i=2}$</th>
<th>$y_{p,i=3}$</th>
<th>$y_{p,i=4}$</th>
<th>$y_{p,i=5}$</th>
<th>...</th>
<th>$y_{p,i=n-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>$y_{0,0}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_1$</td>
<td>$y_{1,0}$</td>
<td>$y_{1,1}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_2$</td>
<td>$y_{2,0}$</td>
<td>$y_{2,1}$</td>
<td>$y_{2,2}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_3$</td>
<td>$y_{3,0}$</td>
<td>$y_{3,1}$</td>
<td>$y_{3,2}$</td>
<td>$y_{3,3}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_4$</td>
<td>$y_{4,0}$</td>
<td>$y_{4,1}$</td>
<td>$y_{4,2}$</td>
<td>$y_{4,3}$</td>
<td>$y_{4,4}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x_5$</td>
<td>$y_{5,0}$</td>
<td>$y_{5,1}$</td>
<td>$y_{5,2}$</td>
<td>$y_{5,3}$</td>
<td>$y_{5,4}$</td>
<td>$y_{5,5}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(...)</td>
<td></td>
</tr>
<tr>
<td>$x_{n-1}$</td>
<td>$y_{n-1,0}$</td>
<td>$y_{n-1,1}$</td>
<td>$y_{n-1,2}$</td>
<td>$y_{n-1,3}$</td>
<td>$y_{n-1,4}$</td>
<td>$y_{n-1,5}$</td>
<td>...</td>
<td>$y_{n-1,n-1}$</td>
</tr>
</tbody>
</table>

Another feature of this method is that the given values of $x_{n-1}$ need not be equally spaced.
To write a program in C++ to interpolate the value of $y_p$ corresponding to a given value of $x_p$ using Aitken’s method, when a set of $n$ tabulated values of $x_i$ and $y_i$ are given.

```c++
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>

void main()
{
    int n, i, j;
    float x[10], y[10][10];

    float xp, yp;
    cout << "Give the no of tabulated data available <=10 :"; cin >> n;
    cout << "Give " << n << " tabulated values of x and f(x): \n";
    for (i = 0; i < n - 1; i++)
        cin >> x[i] >> y[i][0];
    cout << "Give the value of x where f(x) is to be computed: "; cin >> xp;
    for (j = 1; j < n - 1; j++)
    {
        for (i = j; i < n - 1; i++)
        {
            y[i][j] = (y[j - 1][j - 1] * (x[i] - xp) - y[i][j - 1] * (x[j - 1] - xp)) / (x[i] - x[j - 1]);
            cout << endl << "y[" << i << "][" << j << "] = " << y[i][j];
        }
    }
    yp = y[n - 1][n - 1];
    cout << "nThe value of f(x) = " << yp << " corresponding to x = " << xp;
    getch();
}
```
Sample Output

Give the no of tabulated data available <=10 : 6
Give 6 tabulated values of x and f(x):

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>2.4771</td>
</tr>
<tr>
<td>304</td>
<td>2.4829</td>
</tr>
<tr>
<td>305</td>
<td>2.4843</td>
</tr>
<tr>
<td>307</td>
<td>2.4871</td>
</tr>
<tr>
<td>301</td>
<td>2.4786</td>
</tr>
<tr>
<td>306</td>
<td>2.4857</td>
</tr>
</tbody>
</table>

Give the value of x where f(x) is to be computed: 303

y[1][1] = 2.48145
y[2][1] = 2.48142
y[3][1] = 2.48139
y[4][1] = 2.4816
y[5][1] = 2.4814

y[2][2] = 2.48148
y[3][2] = 2.48147
y[4][2] = 2.4815
y[5][2] = 2.48147
y[3][3] = 2.48149
y[4][3] = 2.48149
y[5][3] = 2.48149
y[4][4] = 2.48149
y[5][4] = 2.48149
y[5][5] = 2.48149

The value of f(x) = 2.48149 corresponding to x = 303
INVERSE INTERPOLATION

In some situations, the values of \( f_i \) (for \( i = 0, 1, 2, \ldots, n \)) of the function \( y = f(x) \) at the points \( x_i \) (for \( i = 0, 1, 2, \ldots, n \)) may be given and another point \( x_p \) needs to be determined such that \( f(x_p) = y_p \) when \( y_p \) is given. This is the inverse interpolation or reverse interpolation problem for the function \( f(x) \). The Aitken method can be applied to the inverse interpolation problem by simply interchanging \( x_i \) and \( y_i \) in the corresponding formula (Eq. 5.12).

\[
x_p = x_{i,j} = \frac{x_{j-1,j-1} \times (y_i - y_j) - x_{i,j-1} \times (y_j - y_p)}{y_i - y_{j-1}}
\]

\[
= \frac{1}{y_i - y_{j-1}} \begin{vmatrix}
    x_{j-1,j-1} & y_{j-1} - y_p \\
    x_{i,j-1} & y_i - y_p
\end{vmatrix}
\text{for } i = 1, 2, \ldots, n-1
\]

(5.13)

Inverse interpolation can be done with the help of Lagrange polynomial, when the given points, \( y_i \), are equally spaced. The Lagrange polynomial for inverse interpolation can be written as:

\[
f(y) = f(y_0) \frac{(y - y_1)(y - y_2)\ldots(y - y_n)}{(y_0 - y_1)(y_0 - y_2)\ldots(y_0 - y_n)} + f(y_1) \frac{(y - y_0)(y - y_2)\ldots(y - y_n)}{(y_1 - y_0)(y_1 - y_2)\ldots(y_1 - y_n)}
\]

\[+ f(y_2) \frac{(y - y_0)(y - y_1)(y - y_3)\ldots(y - y_n)}{(y_0 - y_1)(y_2 - y_1)(y_2 - y_3)\ldots(y_2 - y_n)} + \ldots \ldots \]

\[+ f(y_n) \frac{(y - y_0)(y - y_1)\ldots(y - y_{n-1})}{(y_n - y_0)(y_n - y_1)\ldots(y_n - y_{n-1})}
\]

\[
f(y) = \sum_{i=0}^{n} f(y_i) \prod_{j=0}^{n} \left( \frac{y - y_j}{y_i - y_j} \right)
\]

(5.14)

(5.15)
SPLINE INTERPOLATION

- It is one of the most powerful interpolation function
- Commonly used in graphics software
- It has all the characteristics of a good interpolation procedure

Principle of Splines

- The concept of spline is to pass a specified order curve through a pair of points
- We know that we can pass a piece-wise linear curve between two pairs of points
- Can we pass a third order curve between every pair of points?
- The question is how?
- Quadratic spline fits a quadratic between every pair of point and cubic spline fits a cubic between every pair
Curve fitting is the general problem of finding equations of approximating curves which best fit the given set of data.

The general form of a linear equation with one independent variable can be written as $y = a + bx$. 

\[
Y_1 = a_1 x^3 + b_1 x^2 + c_1 x + d_1 \\
Y_2 = a_2 x^3 + b_2 x^2 + c_2 x + d_2
\]

Conditions

\[
Y_1(x_1) = y_1, \quad Y_1(x_2) = y_2 \\
Y_2(x_2) = y_2, \quad Y_2(x_3) = y_3 \\
Y_1'(x_2) = Y_2'(x_2) \\
Y_1''(x_2) = Y_2''(x_2) \\
Y_1''(x_1) = Y_2''(x_3) = 0
\]
where \( a \) and \( b \) are constants (fixed numbers), \( x \) is the independent variable, and \( y \) is the dependent variable. The graph of a linear equation with one independent variable is a straight line, or simply a line.

For a linear equation \( y = a + bx \), the number \( a \) is the \( y \)-value of the point of intersection of the line and the \( y \)-axis. The number \( b \) measures the steepness of the line. \( b \) indicates how much the \( y \)-value changes when the \( x \)-value increases by 1 unit.

**CURVE FITTING WITH A LINEAR EQUATION**

Curve fitting is a procedure in which a mathematical formula (equation) is used to best fit a given set of data points. The objective is to find a function that fits the data overall. Curve fitting is used when the values of the data points have some error, or scatter and require a curve fit to the points.

Curve fitting can be accomplished with many types of functions and with polynomials of various orders.

Curve fitting using a linear equation (first degree polynomial) is the process by which an equation of the form \( y = a + bx \) is used to best fit the given data points. This can be accomplished by finding the constants \( a \) and \( b \).
b that give the smallest error when the data points are substituted in
$y = a + bx$.

![Fig. 6.3: Straight line connecting two points](image1)

![Fig. 6.4: A straight line passing through many data points](image2)

The procedure for obtaining the constants $a$ and $b$ that give the best fit
requires a definition of best fit and an analytical procedure for deriving the
constants $a$ and $b$. The fitting between the given data points and an
approximating linear function is obtained by first computing the error, also
called the residual, which is the difference between a data point and the
value of the approximating function, at each point.

![Fig. 6.5: Curve-fitting points with a linear equation $y = a + bx$](image3)
CRITERIA FOR A “BEST” FIT

A criterion that measures how well the approximating function fits the given data can be determined by computing a total error $E_{in}$ terms of the residuals as

$$E = \sum_{i=1}^{n} e_i = \sum_{i=1}^{n} [y_i - (a + bx_i)]$$

**LINEAR LEAST-SQUARES REGRESSION**

Linear least-squares regression is a method in which the coefficients $a$ and $b$ of a linear function $y = a + bx$ are determined such that the function has the best fit to a given set of data points. The best fit is defined as the smallest possible total error that is computed by adding the squares of the residuals

$$S_r = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - a - bx_i)^2$$

For a given set of $n$ data points $(x_i, y_i)$, the overall error calculated by

$$S_r = \sum_{i=1}^{n} [y_i - (a + bx_i)]^2$$

**LINEAR REGRESSION ANALYSIS**

A regression model is a mathematical equation that describes the relationship between two or more variables. A single regression model includes only two variables: one independent and one dependent. The relationship between two variables in a regression analysis is expressed by a mathematical equation called a regression equation or model. A
regression equation that gives a straight-line relationship between two variables is called a linear regression model; otherwise, it is called a non-linear regression model.

The equation of a linear relationship between two variables $x$ and $y$ is written as $y = a + bx$, where $a$ gives the y-intercept and $b$ represents the slope of the line. In regression model, $x$ is the independent variable and $y$ is the dependent variable. The simple linear regression model for population is written as $y = A + Bx$. This equation is called a deterministic model. It gives an exact relationship between $x$ and $y$.

For the least squares regression line $y = a + bx$
Linear regression provides a powerful technique for fitting a best line to data. There exist many situations in science and engineering that show the relationship between the quantities that are being considered is not linear.

Non-linear regression techniques are available to fit these equations to data directly. A simpler alternative is to use analytical manipulations to transform the equations into a linear form. Then linear regression can be used to fit the equations to data.

\[
b = \frac{SS_{xy}}{SS_{xx}} \quad \text{and} \quad a = \bar{y} - b \bar{x}
\]

\[
SS_{xy} = \Sigma xy - \frac{(\Sigma x)(\Sigma y)}{n}
\]

\[
SS_{xx} = \Sigma x^2 - \frac{(\Sigma x)^2}{n}
\]

\[
SS_{yy} = \Sigma (y - \bar{y}) = \Sigma y^2 - (\Sigma y)^2 / n
\]