# COURSE MATERIAL

SRI CHANDRASEKHARENDRA SARASWATHI VISWA MAHAVIDYALAYA

(Deemed to be University)

Enathur, Kanchipuram-631561

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**BCSF183T40/ BITF183T40**

**Object Oriented Programming Using C++**

Prepared By

*Dr.M.Saraswathi*

*Assistant Professor*

*Dept of CSE*

# Syllabus

| **Course Code :** | **OBJECT ORIENTED PROGRAMMING USING C++** | L T P C |
|---|---|---|
| BCSF183T40 | | 3 0 0 3 |

**PRE-REQUISITE**

Basic Knowledge in C Programming

**OBJECTIVES**

The course will introduce standard tools and techniques for software development, using object oriented approach, use of a version control system, an automated build process, and an appropriate framework for automated unit.

1. To understand the concept of OOP as well as the purpose and usage principles of Inheritance, polymorphism, encapsulation and method overloading
2. To identify classes, objects, members of a class and the relationships among them needed for a specific problem.

**COURSE OUTCOMES:**

After completing the Course, students will learn:

1. Articulate the principles of object-oriented simple abstract data types, control flow and design implementations, using abstraction functions to document them.
2. Outline the essential features of object-oriented programming such as encapsulation, polymorphism, inheritance, and composition of systems based on object identity using class and object.
3. Apply the object using constructors and destructors and using the concept of polymorphism to implement compile time polymorphism in programs by using overloading methods and operators.
4. Use the concept of inheritance to reduce the length of code and evaluate the usefulness.
5. Apply the concept of run time polymorphism by using virtual functions, overriding functions and abstract class in programs.
6. Use I/O operations and file streams in programs and by applying the concepts of class and objects using Generic types , error handling and STL
7. Analyze problems and implement simple C++ applications using an object-oriented programming approach.
8. Name and apply some common object-oriented design patterns and give examples of their use

**UNIT – 1**

Introduction to object oriented programming, Concepts of object oriented programming. C++ programming basics- Data types, Manipulators, Cin, Cout, Type conversion, arithmetic operators, Loops and decisions. Class and objects : Basics of class and objects, access specifiers , member functions defined inside and outside the class.

**UNIT- II**

Constructors and its types, destructors, objects as function arguments, Returning objects from Functions, inline functions, static data and member function. Arrays: Defining & accessing Array elements, arrays as class member data, array of Objects.

**UNIT - III**

Friend functions Friend Classes. Operator Overloading: Overloading Unary Operators, Operator Arguments, Return Values, Overloading Binary Operators – Arithmetic operators, Concatenating Strings, Multiple overloading Comparison operators, Arithmetic Assignment Operators , Overloading the assignment operator

**UNIT- IV**

Inheritance-Derived class and base class, Types of inheritance, derived class constructors, overriding member functions, Public and private inheritance, Class Hierarchies. Memory management -new and delete operator, string class using new. Pointers -Pointers to Objects Referring to Members, Array of pointers to objects.

**UNIT- V**

Virtual Functions, Pure virtual functions, Late Binding, Abstract Classes, Abstract base class , Virtual base classes, the this pointer. Templates - function templates, class template. File Handling-Introduction to graphics. Generic types and collections –Namespace ,error handling , exception handling , signal handling and STL .

**TEXT BOOKS:**

1. Object Oriented Programming in Microsoft C++ - Robert Lafore,Galgotia Publication Pvt Ltd.
2. The Compete Reference C++, Herbert Schlitz, TMH

**REFERENCE BOOKS:**

1. Let us C++ - Yaswant Kanitkar( for templates) ,BPB Publication
2. C++ and Object Oriented Programming Paradigm, PHI
3. C++ : How to Program, 9th Edition, Deitel and Deitel, PHI
4. Object Oriented Programming in C++ - E. Balaguruswamy, Tata Mcgraw Hill.
5. Teach yourself C++ - Herbertsehildt, OSBORNE/MH

**UNIT – 1**

Introduction to object oriented programming, Concepts of object oriented programming. C++ programming basics- Data types, Manipulators, Cin, Cout, Type conversion, arithmetic operators, Loops and decisions. Class and objects : Basics of class and objects, access specifiers , member functions defined inside and outside the class.

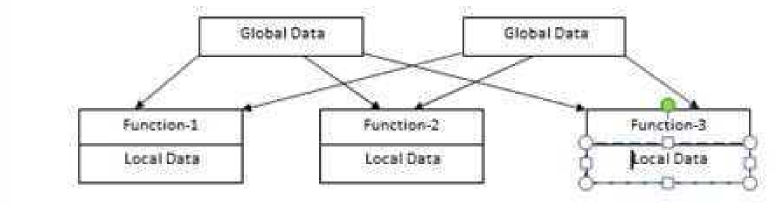
## Introduction to C++

- C++ is an object-oriented programming language
- C++ was developed in 1980 by Bjarne Stroustrup at AT&Tbell Laboratories in MurrayHill,New Jersey, USA.
- C++ is an extension of C with a major addition of the class construct feature.
- Since the class was a major addition to the original C language Stroustrup called the new language 'C with Classes'.
- However later in 1983 the name was changed to C++.
- The idea of C++ comes from the C increment operator ++ thereby suggesting that C++ is an incremented version of C
- C++ is a superset of C.
- The three most important facilities that C++ adds on to C are classes, function overloading, and operator overloading.

**Procedure/ structure oriented Programming**

- Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as procedure-oriented programming (POP).
- In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks.
- The primary focus is on functions.

**Relationship between data and function in pop**



**Introduction to Object Oriented Programming**

- Object oriented programming is the principle of design and development of programs using modular approach.
- Object oriented programming approach provides advantages in creation and development of software for real life application.
- The basic element of object oriented programming is the data.
- The programs are built by combining data and functions that operate on the data.
- Some of the OOP's languages are C++, Java, C #, Smalltalk, Perl, and Python.

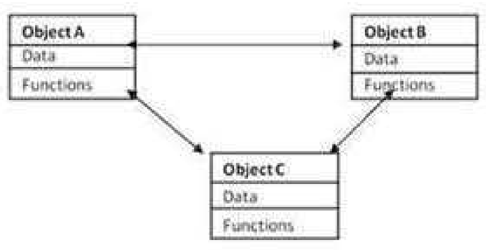
**Definition**

- Object oriented programming (OOP) is a concept that combines both the data and the functions that operate on that data into a single unit called the object.
- An object is a collection of set of data known as member data and the functions that operate on these data known as member function.
- OOP follows bottom-up design technique.
- Class is the major concept that plays important role in this approach. Class is a template that represents a group of objects which share common properties and relationships.

**Features**

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary

**Relationship between data and function in OOP**



| Procedural Programming | Object Oriented Programming |
|---|---|
| Large programs are divided into smaller programs known as functions | Programs are divided into objects |
| Data is not hidden and can be accessed by external functions | Data is hidden and cannot be accessed by external functions |
| Follow top down approach in the program design | Follows bottom-up approach in the program design |
| Data may communicate with each other through functions | Objects may communicate with each other through functions. |
| Emphasize is on procedure rather than data | Emphasize is on data rather than procedure |

**Characteristics of OOP's/Basic Concepts of OOP's**

- Objects
- Class
- Data abstraction
- Data encapsulation
- Inheritance
- Polymorphism
- Message passing
- Dynamic binding

1. **Object**

   *Objects are the basic runtime entities in an object oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.Objects are basic building blocks for designing programs.*

   ***An object is a collection of data members and associated member functions.***

   Example: Apple, orange, mango are the objects of class fruit.

   Each object is identified by a unique name.

   Each object must be a member of a particular class.

2. **Class**

   The objects can be made user defined data types with the help of a class.

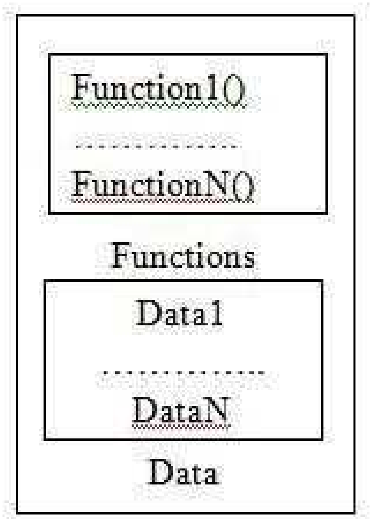
   ***A class is a collection of objects that have identical properties, common behavior and shared relationship.***

   Once class is defined, any number of objects of that class is created.

   Classes are user defined data types.

   A class can hold both data and functions.

   For example: Planets, sun, moon are member of class solar system.



3. **Data abstraction**

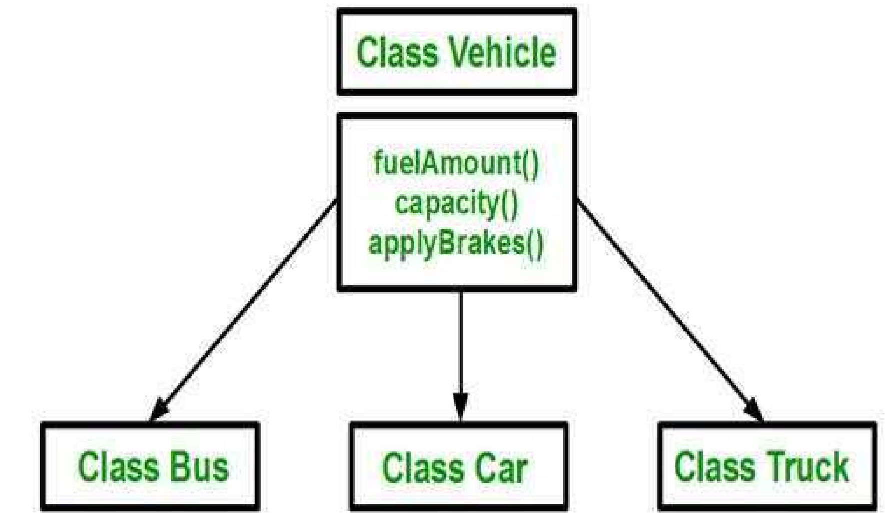
*Data Abstraction refers to the process of representing essential features without including background details or explanations.*

**Data Encapsulation:**

- *The wrapping of data and functions into a single unit (class) is called data encapsulation.*
- Data encapsulation enables data hiding and information hiding.
- *Data hiding is a method used in object oriented programming to hide information within computer* code.

**Inheritance**

- Inheritance is the process by which objects of one class acquire the properties of objects of another class.
- In OOP, the concept of inheritance provides the idea of reusability. This means we can add additional features to an existing class without modifying it.
- The existing class is known as base class or super class.
- The new class is known as derived class or sub class.

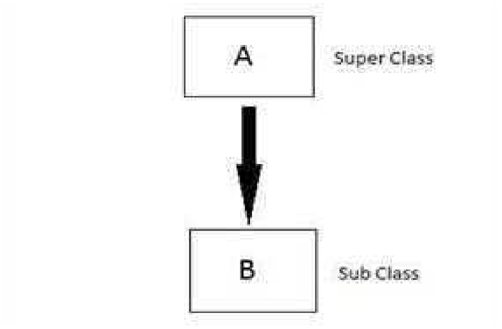


**Types of inheritance:**

1. Single Inheritance
2.Multiple Inheritance
3.Hierarchical Inheritance
4.Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

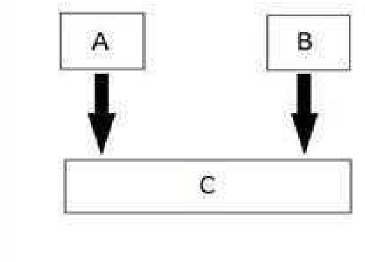
## Single Inheritance

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



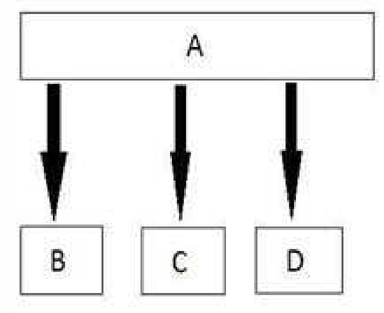
## Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.

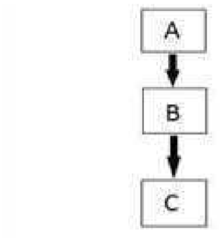


## Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.



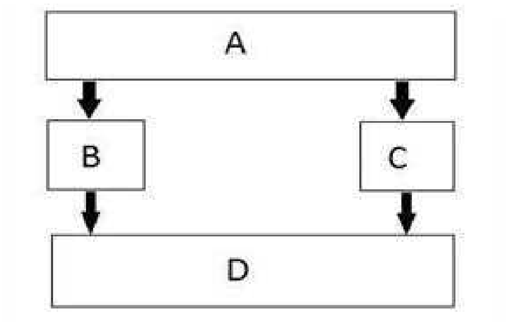
## Multilevel Inheritance



In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.

## Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.

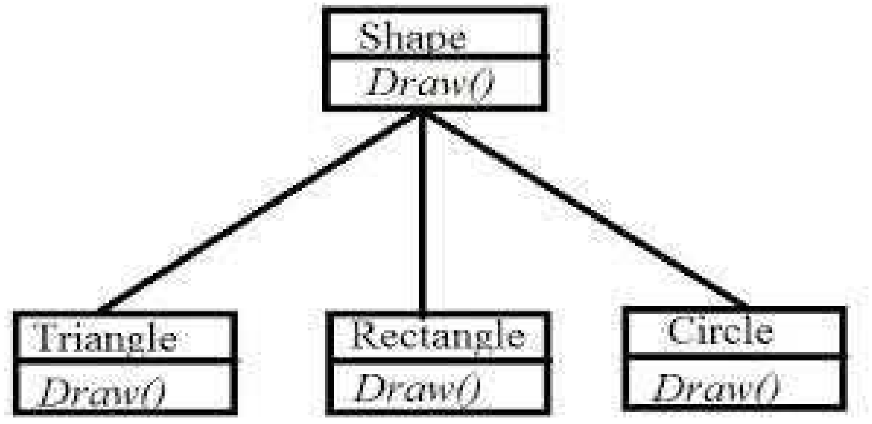


**Polymorphism**

- Polymorphism, a Greek term means to ability to take more than one form.
- An operation may exhibits different behaviors in different instances. The behavior depends upon the type of data used in the operation.
- For example consider the operation of addition for two numbers; the operation will generate a sum. If the operands are string then the operation would produce a third string by concatenation.
- **There are two types of overloading viz.**
  **o Operator Overloading**
  **o Function Overloading**

*The process of making an operator to exhibit different behavior in different instances is known* ***operator overloading.***

***Function overloading*** *means two or more function have same name, but differ in the number of* arguments or data type of arguments.



- Binding is the process of connecting one program to another.

- Dynamic binding is the process of linking the procedure call to a specific sequence of code or function at run time or during the execution of the program.

**Message Passing:**

- In OOP's, processing is done by sending message to objects.
- A message for an object is request for execution of procedure.
- Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.

**Advantages of OOPS:**

- It can implement real time system.
- Software complexity can be reduced.
- It can be easily upgraded to smaller to larger programs.
- Easy to partition the work in project based o objects.
- It supports abstract data types
- Using inheritance, code reusability is possible
- Information hiding can be achieved.

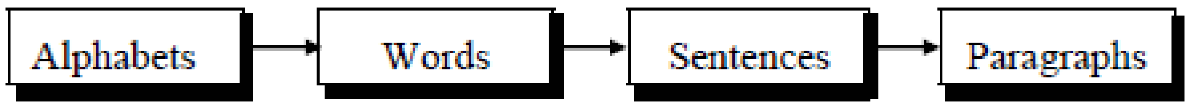
**Applications of oops:**

Computer graphics applications.
- CAD/CAM software
- Object-oriented database.
- User-Interface design such as windows
- Real-time systems.
- Simulation and Modeling
- Artificial intelligence and expert systems.
- Client-Server Systems.

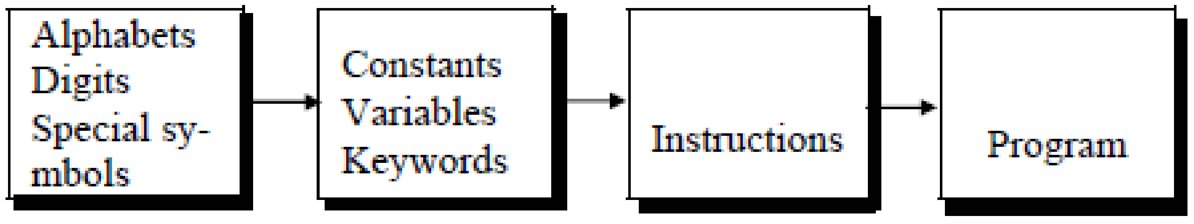
**C++ Programming   Basics**

- Constants
- Variables
- Keywords
- Data types
- Operators
- Control Structures
- Decision Making statements
- Looping Statements
- Manipulators
- Type Conversion

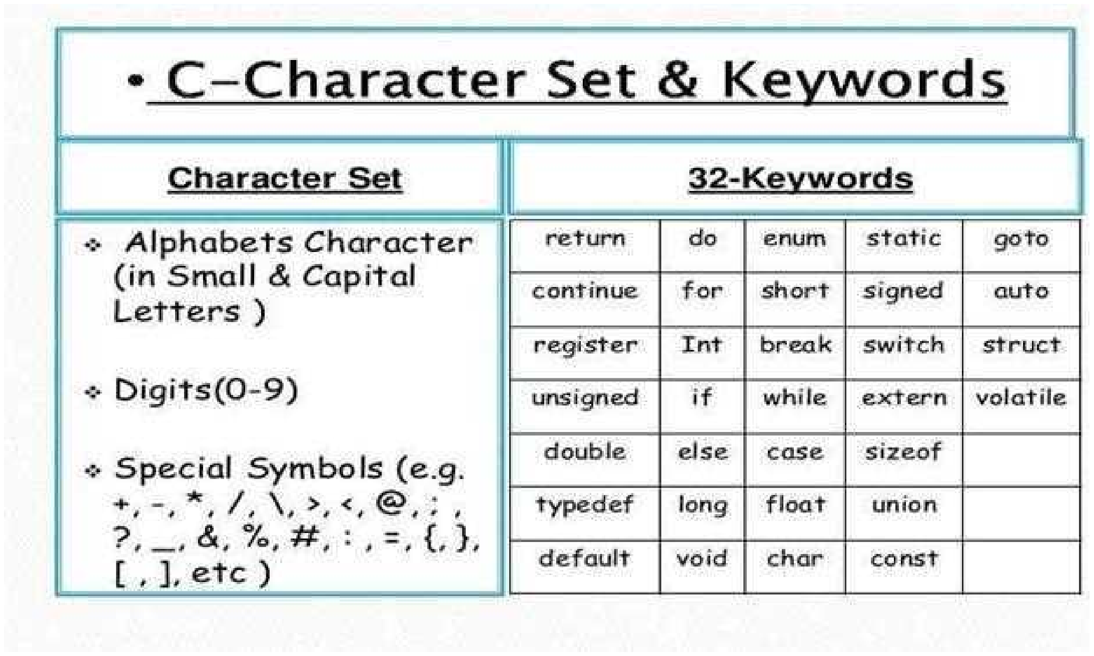
Steps in learning English language:



Steps in learning C:



- The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.
- *Variables* are used to store values that can be changed during the program execution.
    - Int a; /* integer variable */
    - float b; /* float variable */
    - char c; /* char variable */
    - char d[10]; /* String variable */
- *Constant* values that can be changed during the program execution.
    - Int a = 10; 10 is integer constant
    - Float b = 5.5; 5.5 is floating point constant
    - char c=‘a’ a is character constant
    - char d[10]=“scsvmv” scsvmv is string constant
- A Character set denotes any alphabet, digit, or special symbol used to represent information
- "Keywords" are words that have special meaning to the C++ compiler.
- Their meaning cannot be changed at any instance.
- Serve as basic building blocks for program statements.
- All keywords are written in only lowercase.



• C-Character Set & Keywords

| Character Set | 32-Keywords | | | | |
|---|---|---|---|---|---|
| ❖ Alphabets Character (in Small & Capital Letters ) | return | do | enum | static | goto |
| | continue | for | short | signed | auto |
| | register | Int | break | switch | struct |
| ❖ Digits(0-9) | unsigned | if | while | extern | volatile |
| | double | else | case | sizeof | |
| ❖ Special Symbols (e.g. +, -, *, /, \, >, <, @, ; , ?, _, &, %, #, : , =, {, }, [ , ], etc ) | typedef | long | float | union | |
| | default | void | char | const | |

**Rules for Variable declaration**

- May only consist of letters, digits, and underscores
- May not begin with a number
- May not be a C reserved word (keyword)
- Should start with a letter or an underscore(_)
- Can contain letters, numbers or underscore.
- No other special characters are allowed including space.

***Rules for Constants***

Integer Constants

- Refers to sequence of digits such as decimal integer, octal
- integer and hexadecimal integer.
- Some of the examples are 112, 0551, 56579u, 0X2 etc.

Real Constants

– The floating point constants such as 0.0083, -0.78, +67.89

Single Character Constants

– A single char const contains a single character enclosed within

– Pair of single quotes [ ‘ ’ ]. For example, ‘8’, ‘a’ , ‘i’ etc.

• String Constants

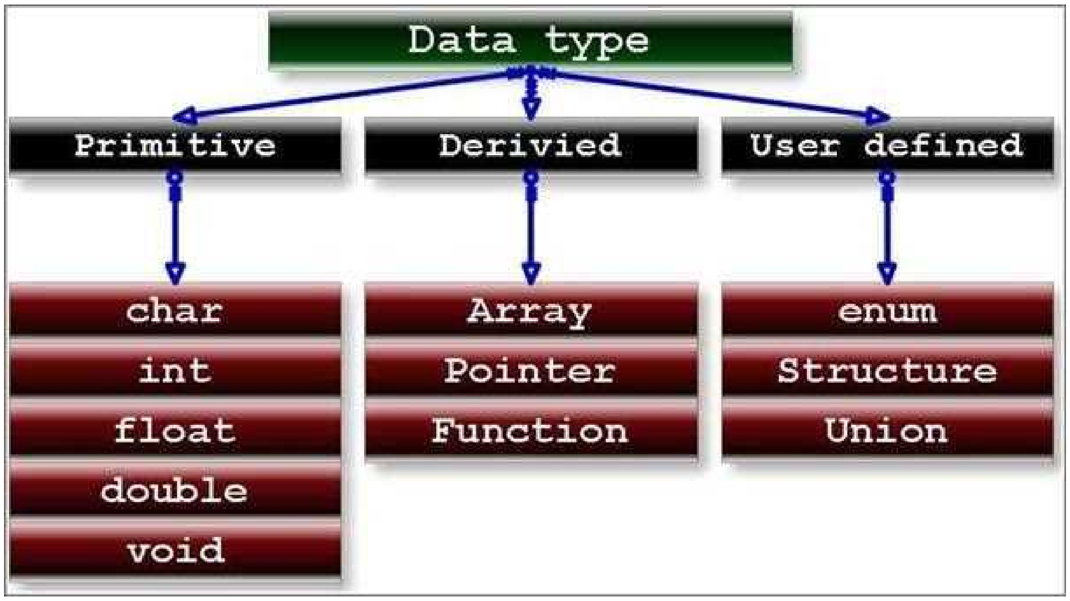
– A string constant is a sequence of characters enclosed in double

– quotes [ “ ” ]; For example, “0211”, “Stack Overflow” etc.

**DATA TYPES:**

–

– Data type is a pre defined keyword which tells compiler what type ofdata a variable can hold



–

## *Declaring Variables*

Before using a variable, you must give the com

The declaration statement includes the data type of the variable. Examples of variable declarations:

int rollno;

Char name[50];

float height ;

Variables are not automatically initialized. For example, after declaration

```
int a;
```

the value of the variable a can be anything (garbage).

## *Initializing Variables*

Initialization of a variable is of two types:

- **Compile time Initialization:** Here, the variable is assigned a value in advance. This variable then acts as a constant.

**Method 1 (Declaring the variable and then initializing it)**

```
int a; a = 5;
```

**Method 2 (Declaring and Initializing the variable together):**

```
int a = 5;
```

**Method 3 (Declaring multiple variables simultaneously and then initializing them separately)**

```
int a, b; a = 5; b = 10;
```

**Method 4 (Declaring multiple variables simultaneously and then initializing them simultaneously)**

```
int a, b;

a = b = 10;

int a, b = 10, c = 20;
```

- **Run time Initialization:** Here, the variable is assigned a value at the run time. The value of this variable can be altered every time the program is being run.

```
int a;

cout<<“enter the value of a”;

cin>>a;
```

**MANIPULATORS**

- Manipulators are used to format the output. C++ has provided some predefined and user defined manipulators. There are number of predefined manipulators present in the header file ‘iomanip.h’.
- Two or more manipulators can be used as a chain in one statement as shown below:

```
cout << manip1  << manip2  << manip3  << item;
cout << manip1 << item1  << manip2 << item2;
```

**1**.endl   manipulator-Is equivalent to \n

Used to move to nextline

2.setw () manipulator:-

▯ Takes integer type of variable as its parameter. The parameter specifies the width of the column.

Example:

```
 cout<< 123<<endl;
cout<< setw (3) << 10;
```

**Output:**

```
123
 10
```

**Example Program using setw and endl manipulators**

```
#include<iostream>
#include<iomanip> // for setw
int main()
{
 int basic = 950, allowance=95, total=1045;
 cout<<setw(10)<<"Basic"<<setw(10)<<basic<<endl;
 cout<<setw(10)<<"Allowance"<<setw(10)<<allowance<<endl;
 cout<<setw(10)<<"Total"<<setw(10)<<total<<endl;
return 0;
}
```

Output of this program is given below.

```
     Basic       950
•  Allowance       95
```

- Total 1045

**3.setprecision () manipulator:**

C++ displays values of float and double type with six digits by default after decimal point.By using the setprecision () manipulator we can pass number of digits we want to display after decimal point.

**Example:**

cout<<setprecision(3);

cout<< sqrt (3) << endl;

**Output**

1.732 (the value of v3 is 1.7320508075689)

**setfill () manipulator:**

- The main task of setfill () manipulator is to fill the extra spaces left in the output of setw() manipulator with characters.

**Example:**

**cout<<setfill('*')**

**cout<<setw (5) << 10;**

cout<<setw (5) << 257<<endl;

**Output: ***10**

****257**

## CIN AND COUT

**cin** and **cout** are standard input and output streams in C++.They are part of the iostream library, which provides manyways to perform input and output operations

Cin – used to take input from user

**Syntax – cin >> variablename;**

Cout – used to print output to display

**Syntax – cout << stringToDisplay/variableName**

## TYPE CONVERSION

**A type cast** is basically a conversion from one type to another. There are two types of type conversion:

Implicit Conversion

Explicit Conversion

**Implicit Conversion**

Done by the compiler on its own, without anyexternal trigger from the user.

Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid loseof data.

All the data types of the variables are upgraded to thedata type of the variable with largest data type**.**

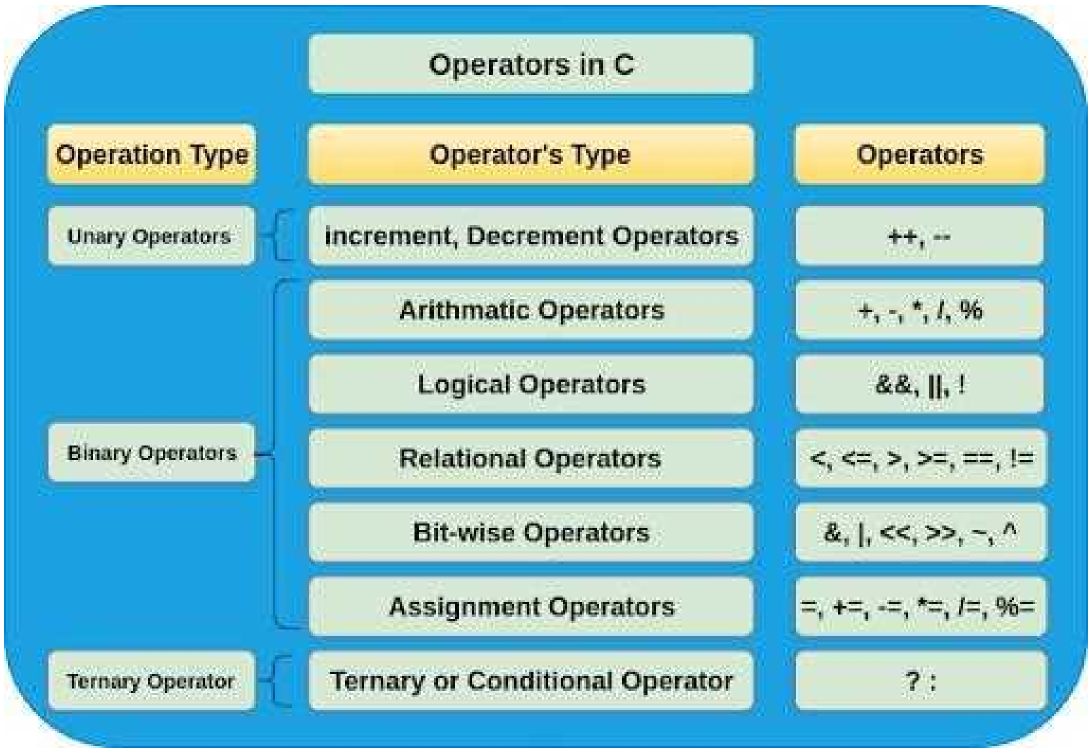
**Explicit Conversion**

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

**Syntax :** (type) expression

```
        float)5;
```

**Operators**



Operators in C

| Operation Type | Operator's Type | Operators |
|---|---|---|
| Unary Operators | increment, Decrement Operators | ++, -- |
| Binary Operators | Arithmatic Operators | +, -, *, /, % |
| | Logical Operators | &&, \|\|, ! |
| | Relational Operators | <, <=, >, >=, ==, != |
| | Bit-wise Operators | &, \|, <<, >>, ~, ^ |
| | Assignment Operators | =, +=, -=, *=, /=, %= |
| Ternary Operator | Ternary or Conditional Operator | ? : |

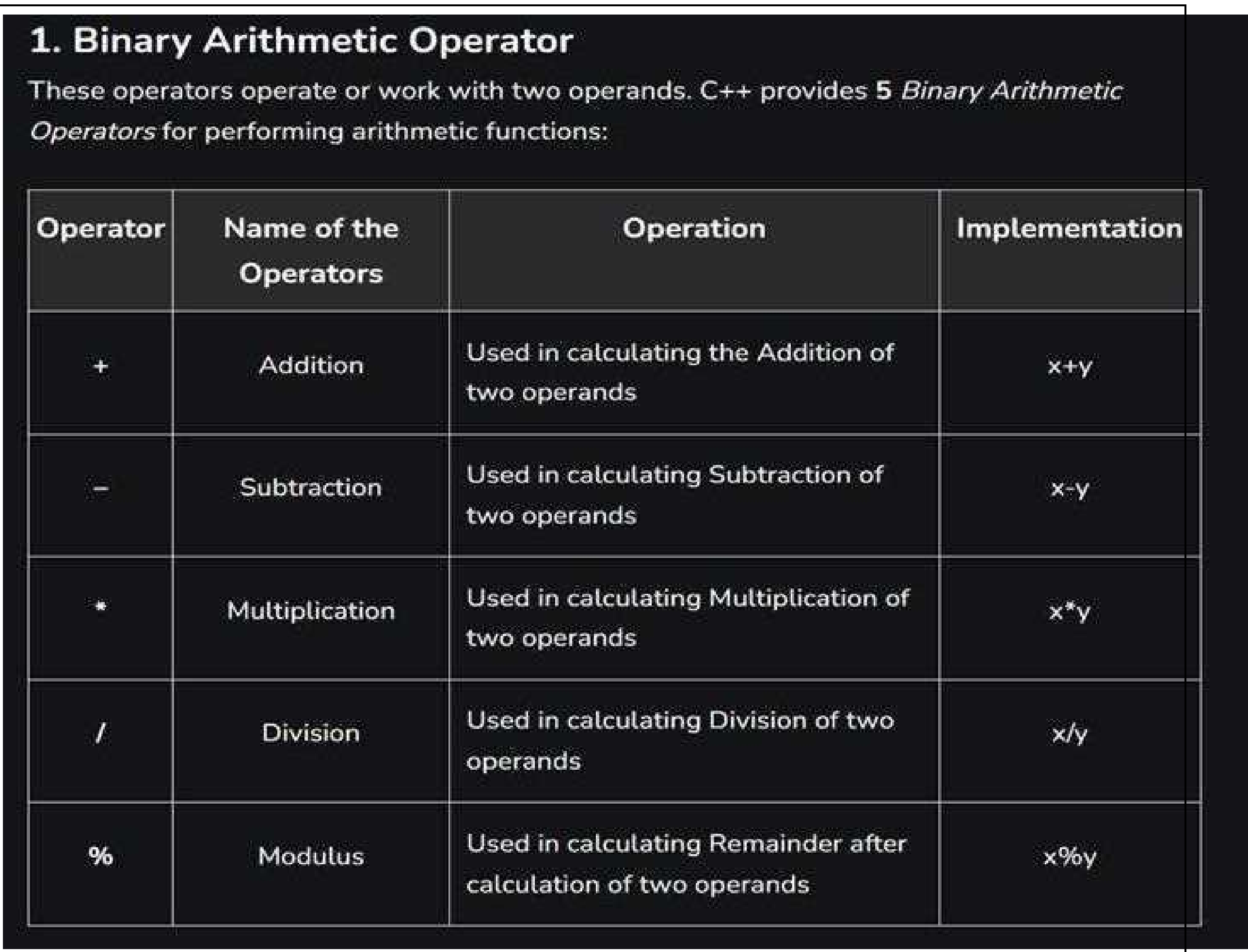
ARITHMETIC OPERATORS

Arithmetic Operators in C++ are used to perform arithmetic or mathematical operations on the operands. For example,
'+' is used for addition, '–' is used for subtraction, '*' is used
for multiplication, etc.

**C++ Arithmetic operators** are of 2 types**:**

Unary Arithmetic Operator

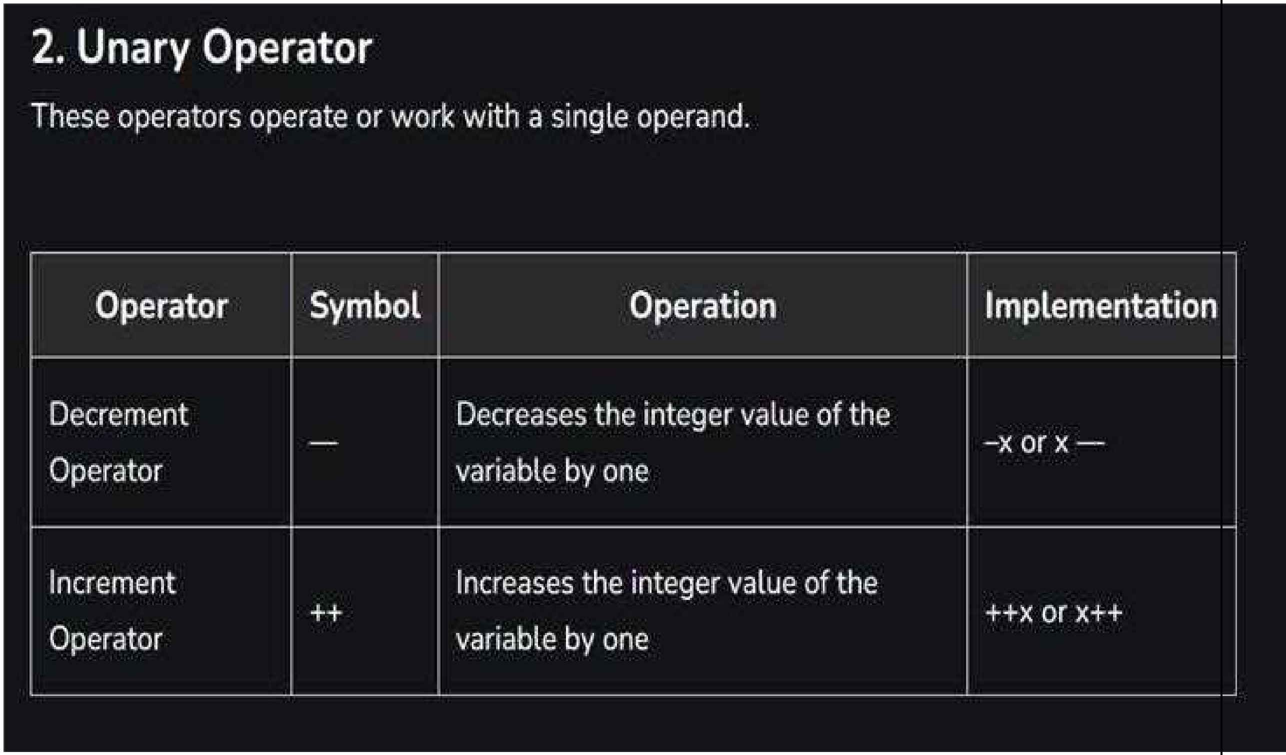
Binary Arithmetic Operator



## 1. Binary Arithmetic Operator

These operators operate or work with two operands. C++ provides **5** *Binary Arithmetic Operators* for performing arithmetic functions:

| Operator | Name of the Operators | Operation | Implementation |
|---|---|---|---|
| + | Addition | Used in calculating the Addition of two operands | x+y |
| – | Subtraction | Used in calculating Subtraction of two operands | x-y |
| * | Multiplication | Used in calculating Multiplication of two operands | x*y |
| / | Division | Used in calculating Division of two operands | x/y |
| % | Modulus | Used in calculating Remainder after calculation of two operands | x%y |



## 2. Unary Operator

These operators operate or work with a single operand.

| Operator | Symbol | Operation | Implementation |
|---|---|---|---|
| Decrement Operator | — | Decreases the integer value of the variable by one | –x or x — |
| Increment Operator | ++ | Increases the integer value of the variable by one | ++x or x++ |

Control Structure

A statement that is used to control the flow of execution in a program is called control structure.

**Types of control structures**

Sequence

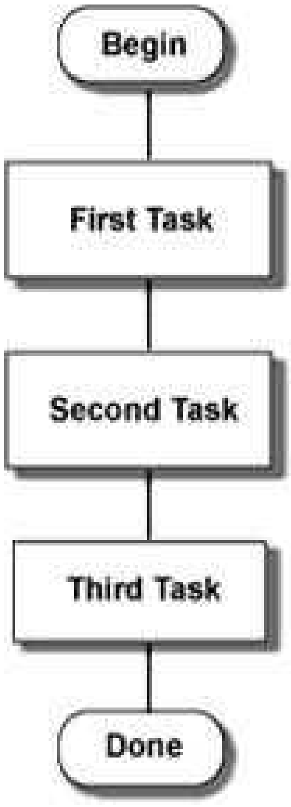
Selection

Repetition

Function call

**Sequence:** Statements are executed in a specified order. No statement is skipped and no statement is executed more than once.

Sequence Structure Flowchart



**For Example**

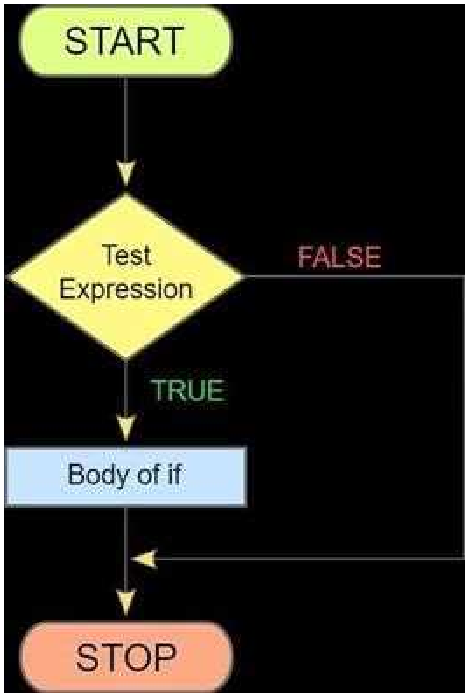
```
#include<iostream.h> void
main()
{
   int a; a=5;
   cout<<"value of  a "<<a;

          }
```

## Selection Structure

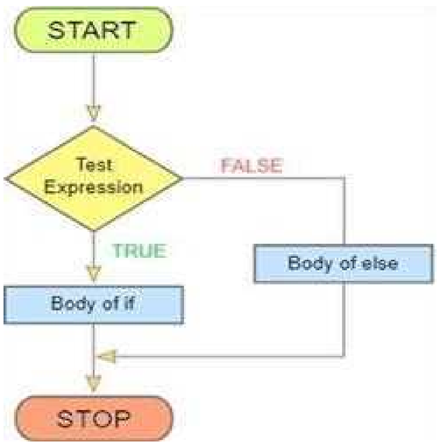
- It selects a statement to execute on the basis of condition.
- Statement is executed when the condition is true and ignored when it is false
- example
  - if,
  - if else,
  - switch structures.

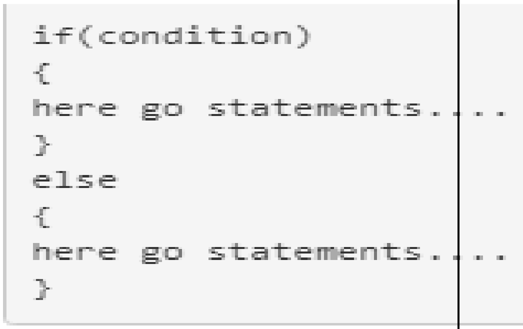
Simple If statement Flow Chart

```
#include <iostream.h> //header file section
#include <conio.h>
Void  main()
{
int age = 18; if(age > 17)
{
cout<<“you are eligible for voting ";
}
cout"\nThis is normal flow ";

}
```



## **if else** Statement Flow Chart





```
if(condition)
{
here go statements....
}
else
{
here go statements....
}
```

**For Example:** #include<iostream.h>
 #include<conio.h> void main ()
{
int y; clrscr();

cout<<"Enter a year:";

cin>>y;

if (y % 4==0)

cout<<“Is a leap year”<<y;

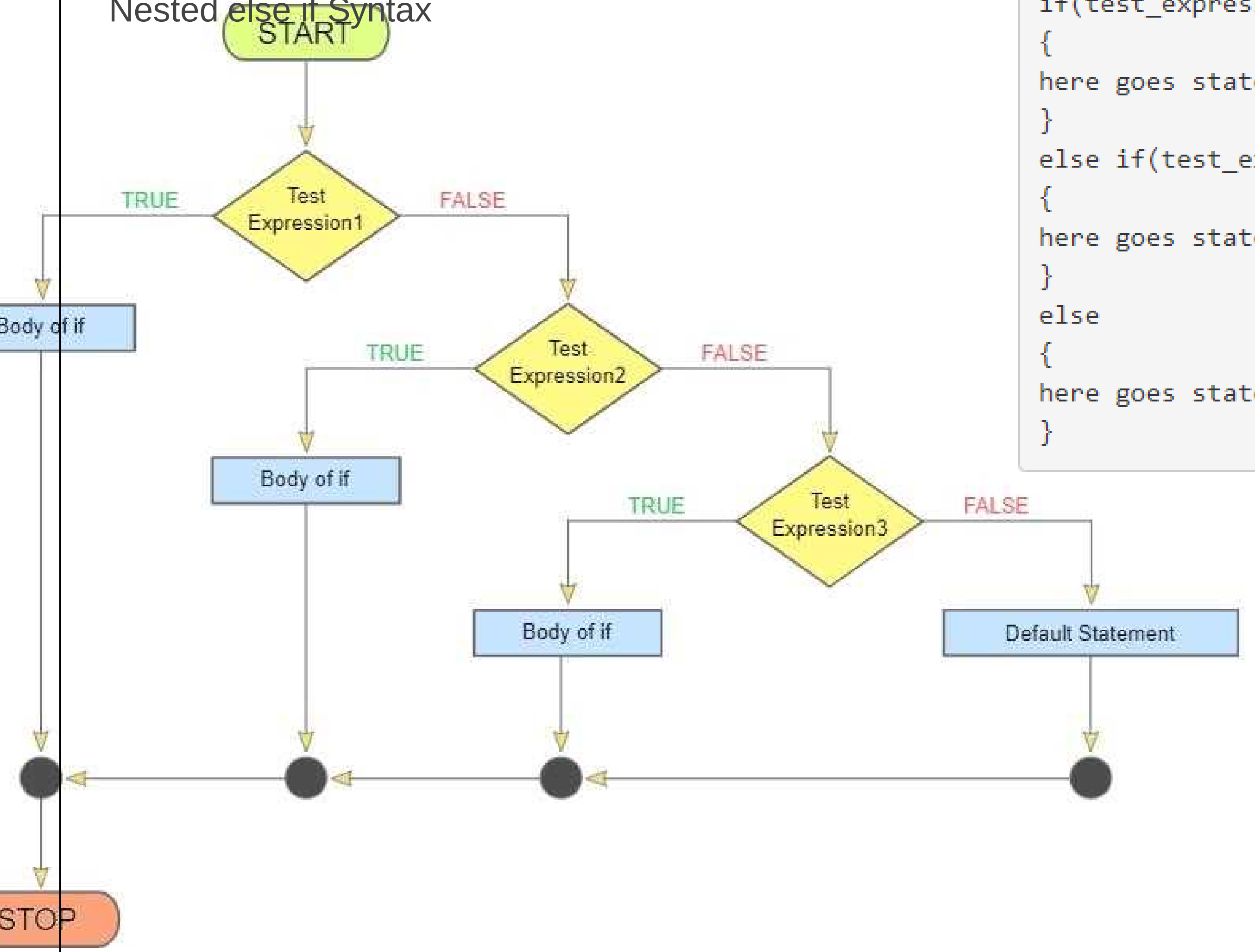
else

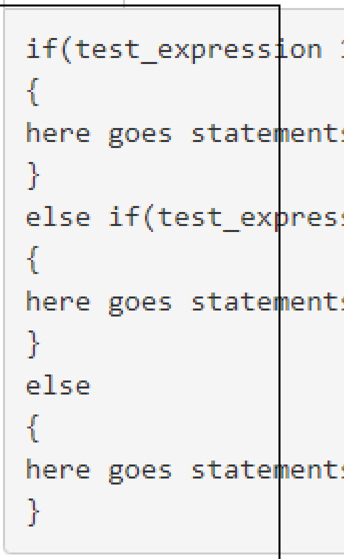
Cout<<“Is not a leap year”<<y;

getch();

}

Nested else if Syntax

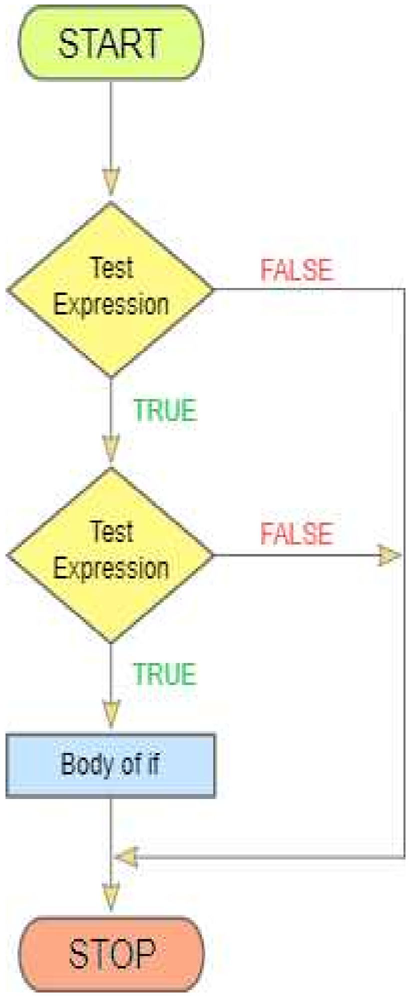


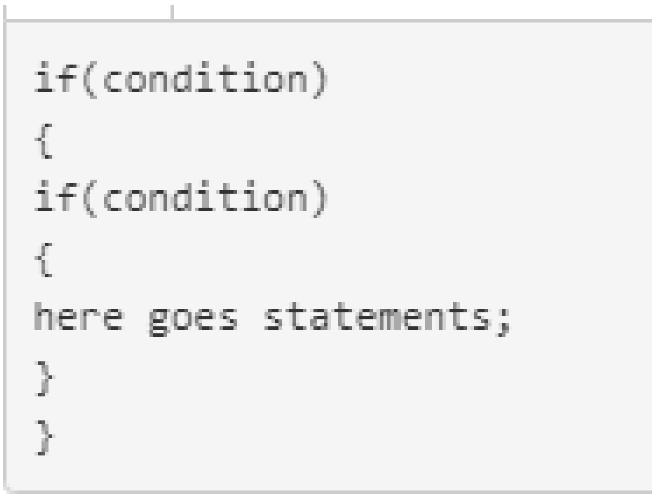

```
if(test_expression 1
{
here goes statements
}
else if(test_expres
{
here goes statements
}
else
{
here goes statements
}
```

**Biggest of three Numbers**

```
#include<iostream.h> #include<conio.h> void main()
{
int a,b,c;
clrscr();
Cout<<"Enter the value of a,b,c"<<\n; Cin>>a>>b>>c;
if((a>b)&&(a>c)) Cout<<"a is greater"<<a; else if(b>c)
Cout<<" b is greater"<<b;
else
Cout<<"c is greater"<<c;
getch();
}
```

Nested if Statement Flowchart



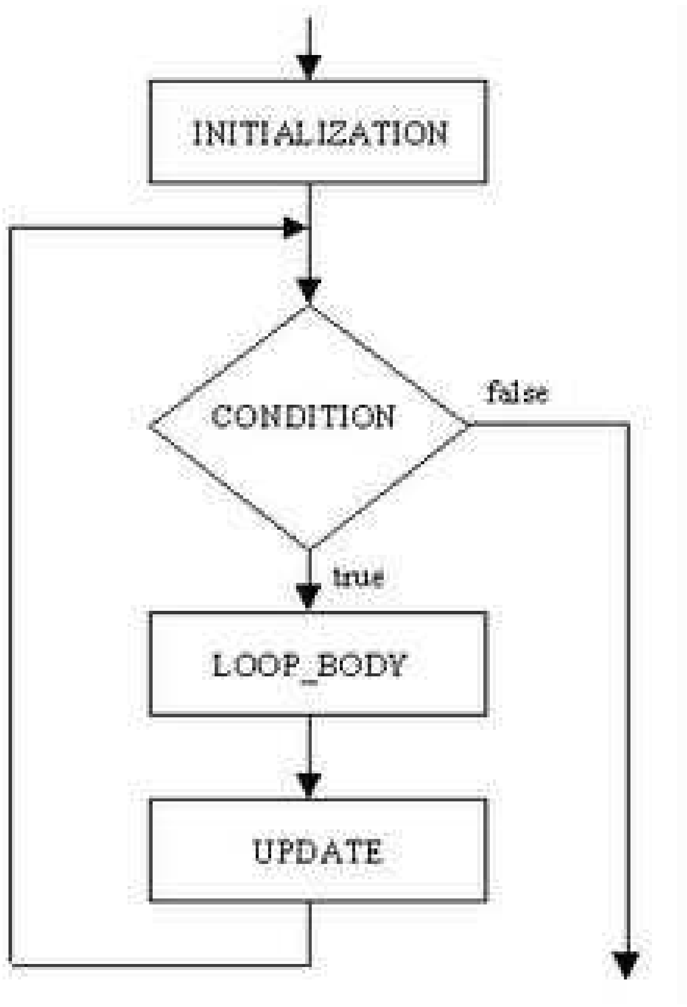


```
if(condition)
{
if(condition)
{
here goes statements;
}
}
```

## Repetition Structure

- In this structure the statements are executed more than one time. It is also known as iteration or loop
- example
  - while loop,
  - for loop
  - do-while loops

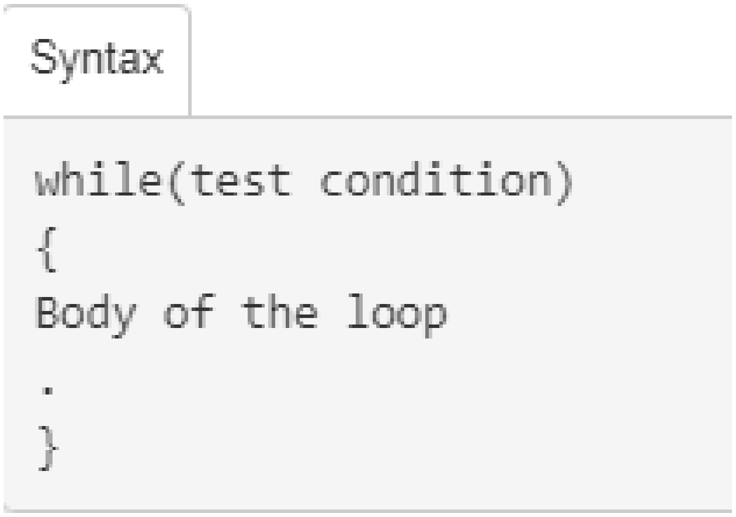
# Repetition Structure Flowchart



## While Loop

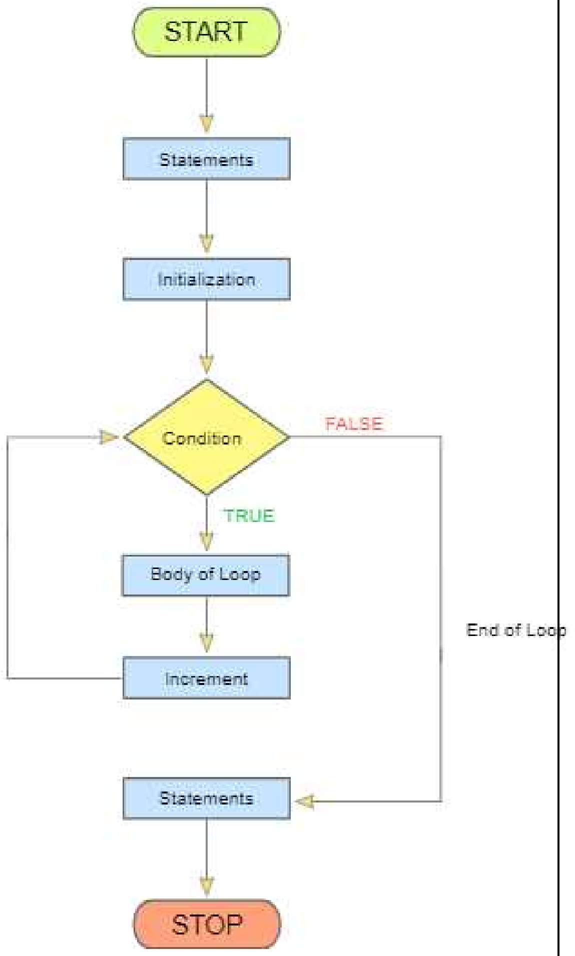
F**or Example:**

```
 #include<iostream.h>
 #include<conio.h>
 void main()
{
int a=1; clrscr(); while(a<=5)
{
cout<<“C++ PROGRAMMING”;
a++;
} //end of while
} //end of main
```



```
while(test condition)
{
Body of the loop
.
}
```

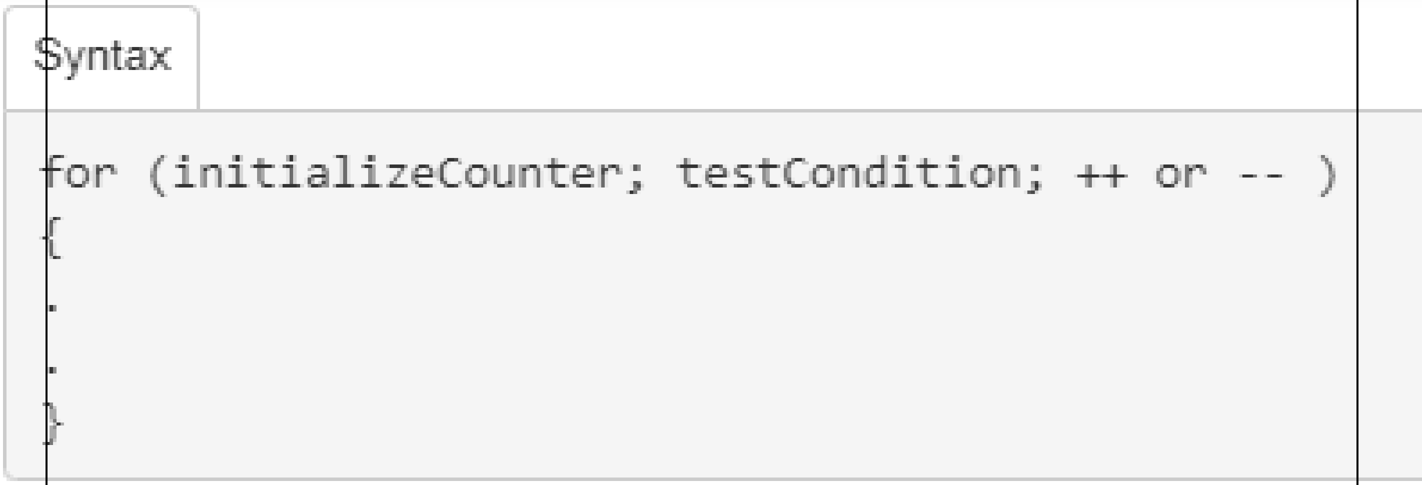
## **while loop** Flowchart



## What is for loop?

Conceptually, **for loop** is a bit more complex than while loop and do while loop though syntax is neat and compact way to write certain types of loops. Typically, a programmer need to use a **for loop** when you exactly know how

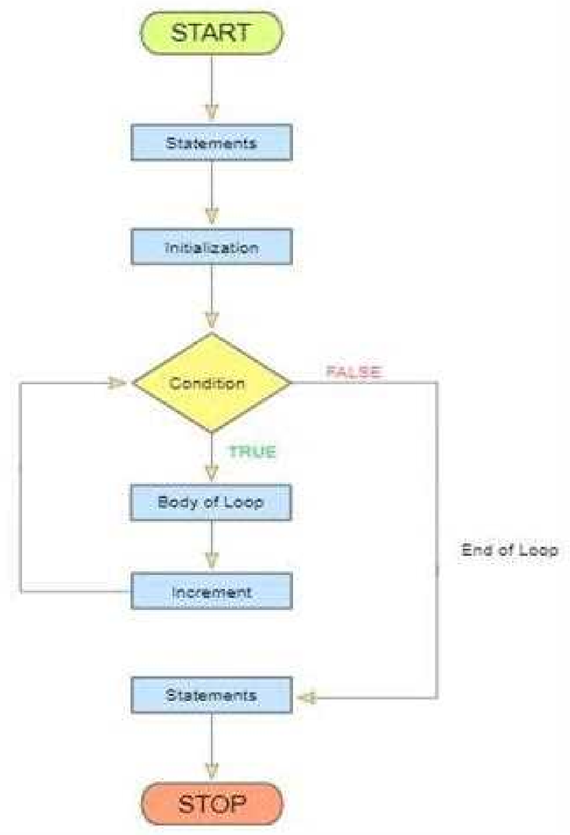
many times you want a block of statement to be executed repeatedly.



Syntax

```
for (initializeCounter; testCondition; ++ or -- )
{
.
.
}
```

## for loop FlowChart



## Example For loop

```
#include<iostream.h>
void main()

{
int n ;
for(n=0; n<10; n++)
{
cout<<”C++ is good”;
}
```

**Program : factorial of given number**

```
#include<iostream.h>
 #include<conio.h> void main ()
{
int i,n;
long int fact=1; clrscr();
cout<<"Enter any number"; Cin>>n;
for(i=1;i<=n;i++)
{
fact=fact*i;
}
Cout<<"Factorial of the given number is “<< fact;
getch();
}
```

output :

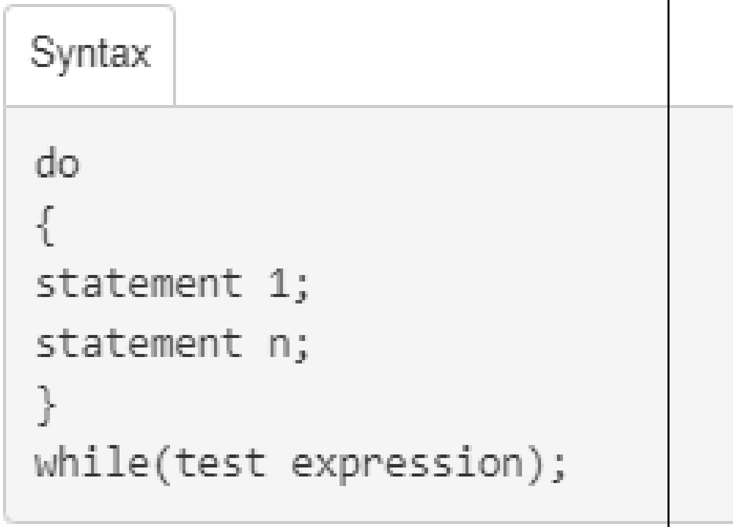
```
Enter any number 6
Factorial of the given number is 720
```

## Do While

The do while loop evaluates the condition only

The statements within the do-while loop executed at least once.

A do while loop is also called as bottom tested loop.


Syntax

```
do
{
statement 1;
statement n;
}
while(test expression);
```

**Note**

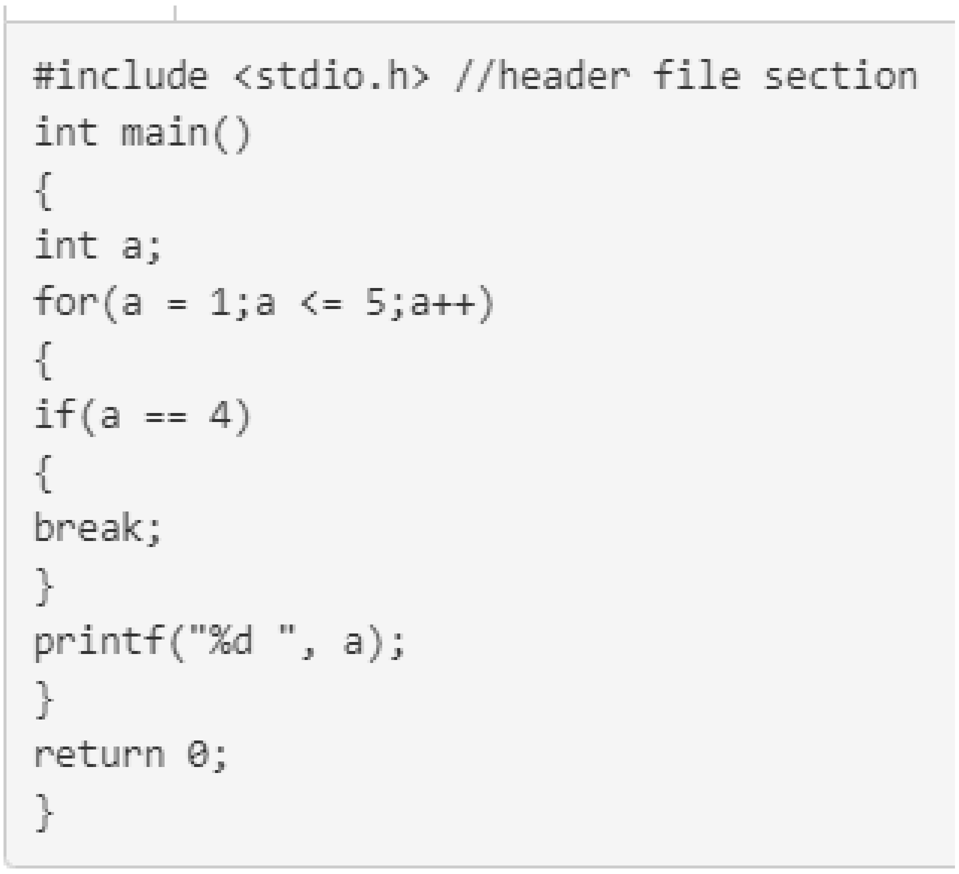
Although the test condition in the while loop may false for the very first time. Th statements of do-while

## Difference while and do..while

| BASIS FOR COMPARISON | W HI LE | DO- WHILE |
|---|---|---|
| General Form | while ( condition) {<br>statements; //body of loop<br>} | do{<br>.<br>statements; // body of loop.<br>.<br>} while( Condition ); |
| Controlling Condition | In 'while' loop the controlling condition appears at the start of the loop. | In 'do-while' loop the controlling condition appears at the end of the loop. |
| Iterations | The iterations do not occur if, the condition at the first iteration, appears false. | The iteration occurs at least once even if the condition is false at the first iteration. |
| Alternate name | Entry-controlled loop | Exit-controlled loop |
| Semi-colon | Not used | Used at the end of the loop |

**Break Statement**

- Though **break statement** comes under decision-making statement, its most oftenly using in looping.
- One can use the break statement to break out from a loop or switch statement.
- When break is executed within a loop, the loop ends immediately, and execution continues with the first statement following the loop.



```
#include <stdio.h> //header file section
int main()
{
int a;
for(a = 1;a <= 5;a++)
{
if(a == 4)
{
break;
}
printf("%d ", a);
}
return 0;
}
```

**CLASS AND OBJECTS:**

**Class and Object**

A class is a way to bind the data and its associated functions together. It allows the data and functions to be hidden, if necessary, from external use. A class declaration is similar syntactically to a structure.

**General form of a class declaration is:**

class classname

{

private:

Variable declaration/data members;

Function declaration/ member functions;

protected:

Variable declaration/data members;

Function declaration/ member functions;

public:

Variable declaration/data members;

Function declaration/ member **functions;**

**};**

**Class Definition – Syntax:**

```
Classs classname{
// member variables

Constructor(){

}
// member functions

};
```

**Object Decleration – Syntax:**

Classname objectName;

If class contains a constructor

Classname objectName();

**ACCESS SPECIFIERS**

**Access modifiers** are used to implement an important aspect of Object-Oriented Programming known as **DataHiding**.

**Public:** Member Functions /Data Members can be accessedfrom anywhere in the program

**Private :** Members functions/Data Members can beaccessed only with in the class

**Protected :** Member Functions/ Data Members can beaccessed only in inherited class.

**Structure of C++ Program**

**General steps to write a C++ program using class and object:**

- Header files
- Class definition
- Member function definition
- void main function

**EX:Write a program to find sum of two integers using class and object.**

**#include<iostream.h>**

```
class Add
{
int x, y, z;
public:
void getdata()
{
cout<<”Enter two numbers”;
cin>>x>>y;
}
void calculate();
void display();
};
void Add :: calculate()
{
z=x+y;
}
void Add :: display()
{
cout<<z;
}
void main()
{
Add a;
a.getdata();
a.calculate();
a.display();
}
```

**A member function can be defined**

**(i)** Inside the class definition

(ii) Outside side the class definition using scope resolution operator (::).

- Here in the above example we are defining the member function getdata() inside the class definition.
- And we are defining the member functions calculate() and display(), outside the class definition using the scope resolution operator.

**Syntax for defining member function outside of the class definition**

**Returntype classname :: memberfunctionname()**
**{**
**}**
**Ex:void add::calculate()**
**{**

**}**

**How to access member of a class?**

To access member of a class dot operator is used. i.e.

object-name.data-member and

object-name.member-function

Application of Scope resolution operator (::)

- It is used to specify scope of a member function.
- It is used to access a global variable.

# UNIT - II

**UNIT- II**

Constructors and its types, destructors, objects as function arguments, Returning objects from Functions, inline functions, static data and member function. Arrays: Defining & accessing Array elements, arrays as class member data, array of Objects.

**Constructors:**

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created.

**Definition:**

A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is the same name as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class. A constructor is declared and defined as follows:

**Characteristics of Constructor:**

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return type, not even void.
- They cannot be inherited, though a derived class can call the base class.
- Like other c++ functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses.
- They make „implicit calls‟ to the operators new and delete when memory allocation is required.

**Example:**

```
Class A
{
   int x,y;
   Public:
   A( )
```

```
            {
            }
A(int a,int b)
{
    x=a;
    y=b;
}
};
Void main ( )
{
   A a;
   a(10,20);
}
```

**Constructors are of 3 types:**

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

**1. Default Constructor:** A constructor that accepts no parameters is called the default constructor.

Ex: item( )

**2. Parameterized Constructors:** The constructors that take parameters are called parameterized constructors.

Ex: item(int a,int b)

**3.Copy Constructor:** A copy constructor is used to declare and initialize an object from another object.

Calling Member Functions :(In Copy Constructor) 2 ways

**Implicit call:** Implicit type conversion is done automatically by the compiler.

```
Ex: a(10,20)          or
    A a(10,20);
```

**Explicit call:** Explicit type conversion is done manually by the programmer.

Ex: A a = A(10,20);

**Constructor Overloading:**

Two or more constructors defined in a single class is called constructor overloading. That means default, parameterized and copy constructor in a single class.

```
class sample
{
   int n;
   Public:
  sample( )   //default constructor
  {
     n=0;
  }
 sample(int a)
 {
     n=a;      //parameterized constructor
 }
Void display( )
{
    cout<<n<<endl;
}
};
Void main( )
{
   sample A(100);       //copy constructors
   sample B(A);
   sample C=A;
   sample D;
   D=A;
   A.display();
   B.display();
   C.display();
   D.display();
}
```

**Output:** 100 100 100 100

**DESTRUCTORS:**

A destructor is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

Eg: ~item() { }

**Characteristics:**

1. A destructor never takes any argument nor does it return any value.
2. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.
3. It is a good practice to declare destructors in a program since it releases memory space for future use.

```
class rectangle
{
  public:
  float length,breadth;
  public:
  rectangle()
  {
   cout<<" \n\n ***** INSIDE THE CONSTRUCTOR ***** \n\n ";
   length = 20;
   breadth = 5;
  }
  public:
  ~rectangle()       //destructor
  {
    cout<<" \n\n ##### INSIDE THE DESTRUCTOR ##### \n\n "<<endl;
  }
};
int main()
{
   clrscr();
  cout<<"PROGRAM FOR CONSTRUCTOR AND DESTRUCTOR"<<endl;
  rectangle rect;
  cout<<"THE LENGTH OF RECTANGLE SET BY CONSTRUCTOR IS:"
         <<rect.length<<endl;
 cout<<"THE BREADTH OF RECTANGLE SET BY CONSTRUCTOR IS:"
     <<rect.breadth<<endl;
return 0;
}
```

**Objects as Function Arguments:**

**Passing an Object as argument:**

To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

**Syntax:**

function_name(object_name);

It can be done in two ways:

* A copy of entire object is passed to function (Pass By Value)

* Only Address of Object Is Passed To Function (Pass By Reference)

**Example**

```
#include<iostream.h>
#include<conio.h>
class distance
{
     int feet,inches;
     public:
     void getdata(int ft,int in)
   {
         feet = ft;
         inches = in;  }
     void sum(distance d1,distance d2)
   {
         inches=d1.inches+d2.inches;
         feet=inches/12;
         inches=inches%12;
```

```
        feet=feet+d1.feet+d2.feet;
    }
     void display()
    {
        cout<<feet<<"FEET"<<endl;
        cout<<inches<<"INCHES"<<endl;
   }
};
  void main()
{
      distance d1,d2,d3;
      clrscr();
      d1.getdata(1,13);
      d2.getdata(3,24);
      d3.sum(d1,d2);
      cout<<" FIRST DISTANCE = ";
      d1.display();
      cout<<"\n SECOND DISTANCE = ";
      d2.display();
      cout<<"\n RESULTANT DISTANCE = ";
      d3.display();
}
```

**RETURNING OBJECTS FROM FUNCTIONS:**

**Syntax:**

object = return object name;

**Example**

```
#include<iostream.h>
#include<conio.h>
class sample
{
     public:
     int d;
     public:
     void add(sample S)
     {
          d = d + S.d;
     }
};
  void main()
  {
       sample S1,S2;
       clrscr();
       S1.d = 50;
       S2.d = 100;
      cout<<" INTIAL VALUES ";
      cout<<" VALUE OF NUMBER 1 :"<<S1.d<<" VALUE OF NUMBER 2:
                  " <<S2.d<<endl;
      cout<<" NEW VALUES  ";
      cout<<"VALUE OF NUMBER 1:"<<S1.d<<" VALUE OF NUMBER 2:"
       <<S2.d<<endl;
       getch();
}
```

**INLINE FUNCTIONS:**

An inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

**Syntax:**

```
inline return-type function-name(parameters)
{
    // function code
}
```

**Advantages:**

1. Function call overhead doesn’t occur.

2.It also saves the overhead of push/pop variables on the stack when a function is called.

3.It also saves the overhead of a return call from a function.

4.An inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function called preamble and return.

**RESTRICTIONS:**

1. If function contains loop (for loop,while loop).
2. If function contains static variables.
3. If function is recursive.
4. If function has return type.
5. If function contains switch or goto statements.

**Example**

```
#include<iostream.h>
#include<conio.h>
inline float mul(float x, float y)
{
    return(x*y);
}
inline double div(double p, double q)
{
    return (p/q);
}
void main()
{
   clrscr();
   float a = 10.5;
   float b = 9.8;
   cout<<" MULTIPLICATION OF "<<a<<" AND "<<b<<" IS "<<endl;
   cout<<mul(a,b)<<endl;
   cout<<" DIVISION OF "<<a<<" AND "<<b<<" IS "<<endl;
   cout<<div(a,b)<<endl;
```

```
    getch();
}
```

**STATIC DATA MEMBERS:**

Static data members are class members that are declared using static keywords.

**Syntax:**

Static int count = 1;

Static data_type member_name**;**

**Outside the class:**

Data_type class_name :: member_name = value;

**Characteristics:**

1. Only one copy is created for entire class and is shared by all objects of class.
2. It is initialized before any object of the class is created, even before the main starts.
3. It is visible only within the class , but its lifetime is entire program.

**STATIC MEMBER FUNCTIONS:**

The functions which can access only static data members are called static member functions. They share a single copy of themselves with different objects of same class.

**Syntax:**

Static return_type function_name(arguments)

**DECLARATION**

**(INSIDE THE CLASS)**

**DEFINITION**

**(OUTSIDE THE CLASS)**

**EX:(with static member function)**

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

```
{
    Private:
    Static int x;       //declaration
    Public:
    Static void function( )
    {
        cout<< " VALUE OF X : "<<x<<endl;
    }
};
    int Demo : : x = 10;     //definition
Void main( )
{
     Demo x;
     x.function( );
     getch( );
}
```

**EX:(without static member function)**

```
#include<iostream.h>
Class demo
{
    Public:
Static int x;
};
Int demo : : x = 10;
Int main( )
{
   cout<<" VALUE OF X : "<<demo : : x;
   getch( );
}
```

**ARRAY:**

Array is a collection of similar data types stored in contagious memory locations (static memory).

**Syntax:**

Data_type array_name[array_size];

**Ex:** int age[20];

*****Array index starts with zero.

**ACCESSING ARRAY ELEMENTS:**

We can access array elements using arr[i] for the particular loop.

**Ex:**

```
for(int i= 0; i<10; i++)
{
    cin>>arr[i] or cin>>age[i];
}
```

**ARRAY AS CLASS MEMBERS :**

By intialising the array in class definition.

**Ex:**

```
Class A
{
    int marks[10];
    char name[20];
}
```

**ARRAY OF OBJECTS:**

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

**Syntax:**

ClassName ObjectName[number of objects];

The Array of Objects stores *objects*. An array of a class type is also known as an array of objects.

**Ex:**

Storing the information of more than one book

```
#include<iostream.h>
#include<conio.h>
class book
{
          protected:
          char name[15];
          int cost;
          public:
          void read();
          void display();
};
     void book :: read()
    {
            cout<< " ENTER THE NAME OF THE BOOK : "<<endl;
            cin>>name;
            cout<<" ENTER THE COST OF THE BOOK : " <<endl;
            cin>>cost;
   }
  void book :: display()
{
        cout<<" #####!!!! BOOK DETAILS !!!!##### "<<endl;
        cout<<" BOOK NAME : " <<name<<endl;
        cout<<" COST : "<<cost<<endl;
}
   void main()
{
       int i,n;
       clrscr();
       book b[3];
```

```
        cout<<"ENTER THE NUMBER OF BOOKS : "<<endl;
        cin>>n;
        for(i=0;i<n;i++)
   {
         b[i].read();
         b[i].display();
   }
    getch();
  }
```

## Unit III

1. Friend functions, Friend Classes.
2. Operator Overloading:

    Overloading Unary Operators, Operator Arguments, Return Values.

    Overloading Binary Operators – Arithmetic operators, Concatenating Strings,

    Multiple overloading Comparison operators, Arithmetic Assignment Operators,

    Overloading the assignment operator.

### FRIEND FUNCTION:

- A Friend function is a non-member function of the class that has been granted access to all private members of the class.
- We simply declare the function within the class by prefixing its declaration with the keyword friend.
- The function can be defined anywhere in the program like a normal C++ function.

### DIFFERENCE BETWEEN MEMBER FUNCTION AND A FRIEND FUNCTION

The major difference between a member function and a friend function is that the member function is accessed through the object while a friend function requires an object to be passed as a parameter.

### CHARACTERISTICS OF FRIEND FUNCTION

- Function definition must not use the keyword **friend or scope resolution operator**.
- The Scope of a Friend function is not inside the class in which it is declared.
- Since its scope is not inside the class, it cannot be called using the object of that class
- It can be called the same as a normal function without using any object.
- It cannot directly access the data members like other member functions and it can access the data members by using an object through the dot operator.
- It can be declared either in the private or public part of the class definition.

- Usually, it has the objects as arguments.

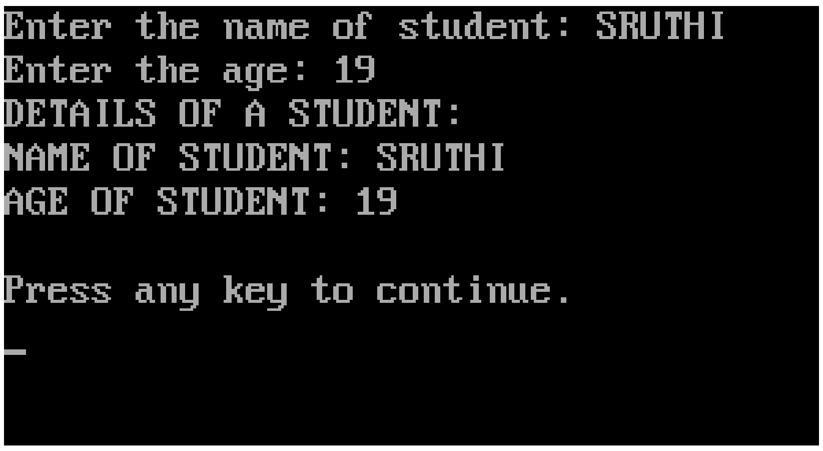
**Syntax:**

```
class ABC
{
 public:
friend void xyz(object of class); //declaring friend function
};
```

**EXAMPLE:**

```
#include<iostream.h>
#include<conio.h>
class student
{
  int age;
  char name[30];
  public:
  void getdata()
  {
    cout<<"Enter the name of student: ";
    cin>>name;
    cout<<"Enter the age: ";
    cin>>age;
  }
  friend void display(student );     //declaring friend function
};
 void display(student a)  //
 {
   cout<<"DETAILS OF A STUDENT: "<<endl;
   cout<<"NAME OF STUDENT: "<< a.name<<endl;
   cout<<"AGE OF STUDENT: "<<a.age<<endl;
 }
void main()
{
  student s;
  clrscr();
  s.getdata();
  display(s);
}
```

**OUTPUT:**



```
Enter the name of student: SRUTHI
Enter the age: 19
DETAILS OF A STUDENT:
NAME OF STUDENT: SRUTHI
AGE OF STUDENT: 19

Press any key to continue.
_
```

**EXAMPLE USING TWO CLASSES:**

```
#include <iostream>
using namespace std;
class TwoValues
{
int a,b;
public:
TwoValues(int i, int j)
 {
a = i;
b = j;
}
friend class Min;  //friend class decalration
};
class Min
{
public:
int minimal(TwoValues  x)
{
return  x.a < x.b ? x.a: x.b;
}
};
int main()
{
TwoValues ob(10, 20);
Min m;
cout << m.minimal(ob);
return 0;
}
```

**OUTPUT:**

```
10
```

**FRIEND CLASS:**

A friend class can access both private and protected members of the class in which it has been declared as a friend. We declared only one function as a friend of another class. But it is possible that all the members of one class can be friend of another class.

**EXAMPLE:**

```
#include <iostream>
using namespace std;
class A
{
    int x =5;
    friend class B;           // friend class.
};
class B
{
  public:
    void display(A &a)
    {
        cout<<"value of x is : "<<a.x;
    }
};
int main()
{
    A a;
    B b;
    b.display(a);
    return 0;
}
```

**OUTPUT:**

```
Value of x is: 5
```

**OPERATOR OVERLOADING:**

In C++, Operator overloading is a compile-time polymorphism.

It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types.

For example, the Addition(+) operator can work on operands of type char, int, float, double, and long int.

However, if s1,s2, and s3 are objects of the class string, then we can write the statement

s3=s1+s2 to concatenate these two objects.

Here, '+' has a special meaning.
Hence, The mechanism of giving special meaning to an operator is known as OPERATOR OVERLOADING.

Operator is a symbol that indicates an operation,

Overloading is assigning different meanings to an operation, depending on the context.

**NOTE: We can overload all C++ operators except the following**

- **Class member access operator( . , *)**
- **Scope resolution operator( :: )**
- **Size operator( size of )**
- **Conditional operator ( ? : )**

## DEFINING OPERATOR OVERLOADING

The general form of an operator function is:

Syntax:

```
return_type   class_name  ::   operator op( argument )
{
     Function body // task defined
}
```

Example:

Int unaryop :: operator +(a)

Here, “operator” is a keyword used for operator overloading.

The operator function should be either a member function or a friend function.

| TYPE | NORMAL FUNCTION | FRIEND FUNCTION |
|---|---|---|
| Unary Operator Overloading | No Arguments | One Argument |
| Binary Operator Overloading | One Argument | Two Arguments |

## TYPES OF OPERATOR OVERLOADING:

There are two types of operator overloading:

1. Unary operator overloading
2. Binary operator overloading

**Unary operator overloading:**

It works only with one class object.

It is the overloading of an operator operating on a single operand.

Ex. unary minus (-), increment( ++ ), decrement (--)

++ or -- can be used as either a prefix or suffix with the function

ex. A=10, a++  =>  A=11

A– => A=9

- (Unary minus)A

10    =>    -10

Unary operator overloading works on only these symbols.

**EXAMPLE:**

**Using normal member functions and declaring the member function outside of the class**

```
#include<iostream.h>

#include<conio.h>

class Unaryop

{

  int x,y,z;

  public:

  Unaryop()

  {

    x=0;

    y=0;

    z=0;

  }

  Unaryop(int a, int b, int c)

  {

   x=a;

   y=b;

   z=c;

  }

  void display()

  {

    cout<<" "<<x<<" "<<y<<" "<<z<<endl;
  }
  void operator - ();
};
void Unaryop :: operator -()
 {
   x=-x;
   y=-y;
   z=-z;
 }
int main()
 {
  Unaryop Un(10,-40,70);
  clrscr();
  cout<<"NUMBERS ARE : "<<endl;
  Un.display();
  -Un;
  cout<<"NUMBERS AFTER OVERLOADED MINUS (-) operator:::  "<<endl;
  Un.display();
  return 0;
 }
```

**OUTPUT:**

```
NUMBERS ARE:
10 -40 70
NUMBERS AFTER OVERLOADED MINUS (-) OPERATOR:::
-10                                   40                                    -70
```

**EXAMPLE FOR ++ and -- OPERATORS:**

```
#include<iostream.h>
#include<conio.h>
class binaryop
{
  int x,y;
  public:
  void get()
  {
      cout<<"Enter two numbers: ";
      cin>>x>>y;
  }
  void display()
  {
    cout<<" "<<x<<" "<<y<<endl;
  }
  void operator ++ ()
  {
   x=++x;
   y=++y;
  }
 void operator --()
  {
     x=--x;
     y=--y;
  }
};
int main()
 {
  binaryop bn;
  clrscr();
  bn.get();
  bn.display();
  bn++;
  cout<<"NUMBERS AFTER OVERLOADING USING ++ operator:::  "<<endl;
  bn.display();
  bn--;
  cout<<"NUMBERS AFTER OVERLOADING USING -- operator:::  "<<endl;
  bn.display();
  return 0;
 }
```

**OUTPUT:**

```
Enter Two Numbers: 4 5
 4 5
Numbers After Overloading Using ++ Operator:::
 5 6
Numbers After Overloading Using -- Operator:::
 4 5
```

**USING FRIEND FUNCTION:**

```
#include<iostream>
using namespace std;
class Unaryop
{
    int a=10;
    int b=20;
    int c=30;
    public:
        void get()
        {
            cout<<"Values of A, B & C"<<endl;
            cout<<a<<" "<<b<<" "<<c<<" "<<endl;
        }
        void show()
        {
            cout<<a<<" "<<b<<" "<<c<<" "<<endl;
        }
        void friend operator-(Unaryop &x);      //Pass by reference
};
void operator-(Unaryop &x)
{
    x.a = -x.a;     //Object name must be used as it is a friend function
    x.b = -x.b;
    x.c = -x.c;
}
int main()
```

```
{
     Unaryop un;
     un.get();
     cout<<"Before Overloading"<<endl;
     un.show();
     cout<<"After Overloading "<<endl;
     -un;
      un.show();
      return 0;
}
```

**OUTPUT:**

```
Values of A, B & C
10 20 30
Before Overloading
10 20 30
After Overloading
-10 -20 -30
```

**Binary Operator Overloading**

A binary operator + can be overloaded to add two objects rather than adding two variables.

Using binary operator overloading a functional notation, C = sum(A, B);can be replaced by, C = A + B;

There should be one argument to be passed. It is the overloading of an operator operating on two operands.

The friend operator function takes two parameters in a binary operator.

The operator function is implemented outside of the class scope by declaring that function as the friend function.

Example:

```
#include<iostream.h>

#include<conio.h>

class complex

{

  double real;

  double img;

  public:

  complex () {}

  complex(double r ,double i)

{

  real = r;

  img = i;

}

complex operator + (complex param)

{

  complex temp;

  temp.real = real + param.real;

  temp.img  = img  + param.img;

  return(temp);

}

  void print()

{

  cout<<real<<" + i "<<img<<endl;

}

};
```

```
int main()
{
  complex c1(3.1,1.5);
  complex c2(1.2,2.2);
  complex c3;
  clrscr();
  c3=c1+c2;
  cout<<"THE COMPLEX NUMBERS ARE: "<<endl;
  c1.print();
  c2.print();
  cout<<"THE ADDITION OF NUMBERS: "<<endl;
  c3.print();
  return 0;
}
```

**OUTPUT:**

```
THE COMPLEX NUMBERS ARE:
3.1 + i 1.5
1.2 + i 2.2
THE ADDITION OF NUMBERS:
4.3 + i 3.7
```

**USING FRIEND FUNCTION:**

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

```
{
   private:
      int real;
      int img;
   public:
      Complex (int r = 0, int i = 0)
      {
         real = r;
         img = i;
      }
      void Display ()
      {
         cout << real << "+i" << img;
      }
      friend Complex operator + (Complex c1, Complex c2);
};

Complex operator + (Complex c1, Complex c2)
{
   Complex temp;
   temp.real = c1.real + c2.real;
   temp.img = c1.img + c2.img;
   return temp;
}

int main ()
```

```
{
    Complex C1(5, 3), C2(10, 5), C3;
    C1.Display();
    cout << " + ";
    C2.Display();
    cout << " = ";
    C3 = C1 + C2;
    C3.Display();
}
```

**OUTPUT**

```
5+i3 + 10+i5 = 15+i8
```

**STRING CONCATENATION USING BINARY OPERATOR OVERLOADING**

Using binary operator overloading, we can concatenate two strings with the “+” operator.

EXAMPLE:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class string
{
  char str[20];
  public:
  void get()
  {
   cout<<" ENTER A STRING: ";
   cin>>str;
  }
  void display()
  {
   cout<<"STRING: "<<str;
  }
  string operator+(string s)
  {
  string obj;
  strcat(str,s.str);
  strcpy(obj.str,str);
  return obj;
  }
};
void main()
{
  clrscr();
  string str1,str2,str3;
  str1.get();
  str2.get();
  str3=str1+str2;
  str3.display();
  getch();
}
```

**OUTPUT:**

```
Enter string: Radhey
Enter string: Krishna
RadheyKrishna
```

**NOTE:**

The operators that cannot be overloaded using the **friend function**.

- Assignment operator ( = )
- Function call operator ( )
- Subscripting operator [ ]
- Class member access operator ( -> ) this pointer

**COMPARISON OPERATORS:**

The comparison operators are very simple.Comparison Operators == and !=

EXAMPLE:

```
#include<iostream>
using namespace std;
class Equal
{
   private :
      int number;
   public :
      void get()
      {
         cin>>number;
      }
      int operator ==  (Equal x)
      {
            if(number==x.number)
            return 1;
            else
            return 0;
        }
        int operator !=  (Equal x)
        {
            if(number!=x.number)
            return 1;
            else
            return 0;
        }
};
int main()
{
    Equal n1,n2;
    cout<<"Please  enter 1st number.  ";
    n1.get();
    cout<<"Please  enter 2nd number. ";
    n2.get();
    if(n1==n2)
    {   cout<<"n1 is equal to n2. ";}
    else if(n1!=n2)
    {   cout<<"n1 is not equal to n2. ";}
    else
    {   cout<<"Sorry "; }
 return 0;
}
```

```
OUTPUT:
Please  enter 1st number.  34
Please  enter 2nd number. 23
n1 is not equal to n2.
```

**ARITHMETIC ASSIGNMENT OPERATORS:**

```
#include<iostream>
using namespace std;
class Distance
{
private:
int feet;
float inches;
public:
Distance(){
feet=0, inches=0;
}
Distance(int ft, float in)
{
feet=ft;
inches=in;
}
void getdist()
{
cout <<" —\nEnter feet: ";
cin >> feet;
cout << "—Enter inches: ";
cin >> inches;
}
```

```
void showdist()
{
cout << feet << " "<< inches <<" ";
}
void operator += ( Distance );
};

void Distance::operator += (Distance d2)
{
feet += d2.feet;
feet=feet+d2.feet;
inches += d2.inches;
if(inches >= 12.0)
{
inches -= 12.0;
feet++;
}
}
int main()
{
Distance dist1;
dist1.getdist();
cout <<" dist1 = ";
dist1.showdist();
cout<<endl;
Distance dist2(11, 5);
cout << "dist2 = ";
dist2.showdist();
cout<<endl;
```

```
dist1 += dist2;
cout << "after addition: ";
cout << "dist1 = ";
dist1.showdist();
cout << endl;
return 0;
}
```

OUTPUT:

```
Enter feet: 20
Enter inches: 5
dist1 = 20 5
dist2 = 11 5
after addition: dist1 = 42 10
```

**ASSIGNMENT OPERATOR OVERLOADING:**

You can overload the assignment operator (=) just as you can other operators.

EXAMPLE:

```
#include <iostream>
using namespace std;
class Distance
{
  private:
    int feet;            // 0 to infinite
    int inches;          // 0 to 12
  public:
    Distance()
   {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i)
    {
      feet = f;
      inches = i;
    }
    void operator = (const Distance &D ) {
      feet = D.feet;
      inches = D.inches;
    }
    void displayDistance()
   {
      cout << "Feet: " << feet <<  " Inches:" <<  inches << endl;
    }
};

int main()
{
  Distance D1(11, 10), D2(5, 11);
  cout << "First Distance: ";
  D1.displayDistance();
  cout << "Second Distance:";
  D2.displayDistance();
  // use assignment operator
  D1 = D2;
  cout << "First Distance: ";
  D1.displayDistance();
  return 0;
```

```
}
```

**OUTPUT:**

```
First Distance : Feet: 11 Inches: 10

Second Distance :Feet: 5 Inches: 11

First Distance :Feet: 5 Inches: 11
```

## UNIT- IV

Inheritance-Derived class and base class, Types of inheritance, derived class constructors, overriding member functions, Public and private inheritance, Class Hierarchies. Memory management -new and delete operator, string class using new. Pointers -Pointers to Objects Referring to Members, Array of pointers to objects.

**Definition:-**

Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.

The class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

**Advantages of inheritance:-**

Reduced Code Duplication: Inheritance reduces code duplication by allowing derived classes to reuse the properties and methods defined in the base class.
Developers don't have to write the same code multiple times in different classes when a common functionality exists.

**Derived class:-** A Derived class is defined as the class derived from the base class.

**The Syntax of Derived class:**

```
class derived_class_name :: visibility-mode base_class_name

{

   //body of derived class

}
```

**Visibility modes:-**

- Since the member ‘x’ in A is **public**, its visibility will be open to all. It means that any class can access and use this x. That is the reason there is no error in ‘b.x’.

- The member ‘y’ in A is **protected**, The its visibility will be only to the derived class. It means that any derived class can access and use this y.
- The member ‘z’ in A is **private**, its visibility will not be open to any other class. It means that any derived class cannot access and use this z.

**Public Visibility mode:** If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.

**Protected Visibility mode:** If we derive a subclass from a Protected base class. Then both public member and protected members of the base class will become protected in the derived class.

**Private Visibility mode:** If we derive a subclass from a Private base class. Then both public member and protected members of the base class will become Private in the derived class.

**Types of Inheritance:-**

1.Single inheritance

2.Multiple inheritance

3.Multilevel inheritance

4.Hierarchical inheritance

5.Hybrid inheritance

**Single Inheritance:-**

When one class inherits another class, it is known as single level inheritance.

```
#include<iostream.h>

#include<conio.h>

class person

{

  private:

  int age;
```

```
   public:
   void getage()
   {
     cout<<" ENTER THE AGE : "<<endl;
     cin>>age;
   }
   void show_age()
   {
      cout<<" AGE : "<<age<<endl;
   }
};
class aim : public person
{
   private:
   int salary;
   public:
   void getsalary()
   {
     cout<<"  ENTER THE SALARY :  "<<endl;
     cin>>salary;
   }
   void show_salary()
   {
      cout<<" SALARY : "<<salary<<endl;
   }


};
void main()
{
  aim a;
  clrscr();
  a.getage();
  a.getsalary();
  a.show_age();
  a.show_salary();
getch();
}
```

**Multiple Inheritance:-**

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

```
#include<iostream.h>
#include<conio.h>
class car
{
  private:
  char name1[20],name2[20];
  public:
  void getdata()
  {
    cout<<" ENTER THE NAME OF CAR 1 : "<<endl;
    cin>>name1;
    cout<<" ENTER THE NAME OF CAR 2 : " <<endl;
    cin>>name2;
  }
  void display()
  {
    cout<<"1.CAR : "<<name1<<endl;
    cout<<"2.CAR : "<<name2<<endl;
  }
};
class city
{
   private:
   char city1[20],city2[20];
   public:
   void getcity()
   {
     cout<<" ENTER CITY ONE : "<<endl;
     cin>>city1;
     cout<<" ENTER CITY TWO : "<<endl;
     cin>>city2;
   }
   void showcity()
   {
     cout<<" 1.CITY : "<<city1<<endl;
     cout<<" 2.CITY : "<<city2<<endl;
   }
};
class brand : public car , public city
{
  private:
  char brand1[10],brand2[10];
  public:
  void getbrand()
  {
    getdata();
    getcity();
    cout<<" ENTER BRAND 1 : "<<endl;
    cin>>brand1;
    cout<<" ENTER BRAND 2 : "<<endl;
    cin>>brand2;
  }
  void showbrand()
  {
    display();
    showcity();
    cout<<" 1.BRAND : "<<brand1<<endl;

    cout<<" 2.BRAND : "<<brand2<<endl;

  }

};

void main()

{

  brand b;

  clrscr();

  b.getbrand();

  b.showbrand();

getch();

}
```

**Multilevel Inheritance:-**

Multilevel inheritance is a process of deriving a class from another derived class.

```
#include<iostream.h>

#include<conio.h>

class student

{

   char name[10];

   int age;

   public:

   void getdata()

   {

     cout<<" ENTER THE NAME OF THE STUDENT : "<<endl;
     cin>>name;
     cout<<" ENTER THE AGE OF THE STUDENT : "<<endl;
     cin>>age;
   }
   void display()
   {
      cout<<" NAME : "<<name<<endl;
      cout<<" AGE : "<<age<<endl;
   }
};
class test : public student
{
  protected:
  int math,eng,sci;
  public:
  void gettest()
  {
     getdata();
     cout<<" ENTER MATHS MARKS : ";
     cin>>math;
     cout<<" ENTER ENGLISH MARKS : ";
     cin>>eng;
     cout<<" ENTER SCIENCE MARKS : ";
     cin>>sci;
  }
  void show_test()
  {
     display();
     cout<<" MATH MARKS : "<<math<<endl;
     cout<<" ENGLISH MARKS : "<<eng<<endl;
     cout<<" SCIENCE MARKS : "<<sci<<endl;
  }
};
class result : public test
{
   protected:
   int total,avg;
   public:
   void getresult()
   {
      gettest();
      total = math+eng+sci;
      avg=total/3;
   }
   void show_result()
   {
     show_test();
     cout<<" TOTAL : "<<total<<endl;
     cout<<" AVERAGE : "<<avg<<endl;
   }
};

void main()
{
    result r;
    clrscr();
    r.getresult();
    r.show_result();
getch();
}
```

**Hierarchial Inheritance:-**

It is the hierarchy of classes. There is a single base class and multiple derived classes. Furthermore, the derived classes are also inherited by some other classes. Thus a tree-like structure is formed of hierarchy.

**Hybrid Inheritance:-**

Combination of any of the above types of inheritance.

## Memory management Operators in c++

### malloc() vs new in C++

Both the **malloc()** and new in C++ are used for the same purpose. They are used for allocating memory at the runtime. But, malloc() and new have different syntax. The main difference between the malloc() and new is that the new is an operator while malloc() is a standard library function that is predefined in a **stdlib** header file.

The new is a memory allocation operator, which is used to allocate the memory at the runtime.

**Syntax of new operator**

type  pointer variable = **new** type(parameter_list);

example:

int *p=new int[];

**In the above syntax**

**type:** It defines the datatype of the variable for which the memory is allocated by the new operator.

**variable:** It is the name of the variable that points to the memory.

**parameter_list:** It is the list of values that are initialized to a variable.

**Example**

```
#include <cstring> //for strlen
using namespace std;
int main()
{
Char *str = “Idle hands are the devil’s workshop.”;
int len = strlen(str);
char* ptr = new char[len+1];   //set aside memory:
strcpy(ptr, str);                    //copy str to new memory area ptr
cout << “ptr=” << ptr << endl;
delete ptr;                  //  delete pointervariable
return 0;
```

**The delete Operator**

delete[] ptr;

**A String Class Using new**

```
// using new to get memory for strings
#include <iostream>
#include <cstring>      //for strcpy(), etc
using namespace std;

class String
{
private:
char* str;                    //pointer to string
public:
String(char* s)           //constructor, one arg
{
int length = strlen(s);              //length of string argument
 char str = new char[length+1];                       //get memory
strcpy(str, s);   str=hello world
}
~String() //destructor
{
cout << “Deleting str.\n”;
delete   str;                              //release memory
}
void display()
{
cout << str << endl;
}
};

int main()
{

String s1 = “hello world.”;
cout << “s1=”; //display string
s1.display();
return 0;
}
```

**Pointers to Objects**

Pointers can point to objects as well as to simple data types and arrays. We've seen many examples of objects defined and given a name, in statements like

Distance dist;

where an object called dist is defined to be of the Distance class.

program, ENGLPTR, that compares the two approaches to creating objects.

```
#include <iostream>
using namespace std;
class Distance
{
private:
int feet;
float inches;
public:
void getdist()   //get length from user
{
cout << “\nEnter feet: “;
cin >> feet;                          feet=5
cout << “Enter inches: “;                    inches=4.6
cin >> inches;
}
void showdist()

{ cout << feet << “\’-” << inches << ‘\”’; }  5  -4.6
};
```

```
int main()
{
Distance dist;              //define a named Distance object
dist.getdist();                         //access object members
dist.showdist();                                //  with dot operator
```
**Distance* distptr;                              //pointer to Distance**

**distptr = new Distance;**

```
Distance *distptr=new Distance;
distptr->getdist();  ->              //access object members
distptr->showdist();                  // with -> operator
cout << endl;
return 0;
}
```

**Another Approach to new**

```
#include <iostream>
using namespace std;
class Distance // English Distance class
{
private:
int feet;
float inches;
public:
void getdist() // get length from user
{
cout << “\nEnter feet: “; cin >> feet;
cout << “Enter inches: “; cin >> inches;
}
void showdist() // display distance
{ cout << feet << “\’-” << inches << ‘\”’; }
};
```

```
int main()
{
Distance & dist = *(new Distance); // create Distance object
// alias is “dist”
dist.getdist(); // access object members
dist.showdist(); // with dot operator
cout << endl;
return 0;
}
```

**An Array of Pointers to Objects**

A common programming construction is an array of pointers to objects. This arrangement allows easy access to a group of objects, and is more flexible than placing the objects themselves in an array

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class  person //class of persons
{
protected:
char name[40];
public:
void setName() //set the name
{
cout << “Enter name: “;
cin >> name;
}
void printName() //get the name
{
cout << “\n Name is: “ << name;
}
};
```

```
int main()
{
Person *persPtr[100];

                                                //array of pointers to persons
int n = 0;                            //number of persons in array
char choice;
do
{
persPtr[n] = new person;              //make new object
persPtr[n]->setName();                //set person's name
n++; //count new person
cout << "Enter another (y/n)? ";      //enter another
cin >> choice; //person?
}
while( choice=='y' );                 //quit on 'n'
for(int j=0; j<n; j++)
{
cout << "\nPerson number " << j+1;
persPtr[j]->printName();
}
cout << endl;
return 0;
}
```

UNIT -5

Virtual Functions, Pure virtual functions, Late Binding, Abstract Classes, Abstract base class , Virtual base classes, this pointer.
Templates - function templates, class template. File Handling-Introduction to graphics.
Generic types and collections – Namespace, error handling , exception handling, signal handling and STL.

**POLYMORPHISM**

---

COMPILE - TIME POLYMORPHISM

- FUNCTION OVERLOADING
- OPERATOR OVERLOADING

RUN - TIME POLYMORPHISM

- VIRTUAL FUNCTIONS
- PURE VIRTUAL FUNCTIONS

**VIRTUAL FUNCTIONS :**

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.
- They are mainly used to achieve Runtime polymorphism.
- Functions are declared with a virtual keyword in a base class.
- The resolving of a function call is done at runtime.

**Rules for Virtual Functions**

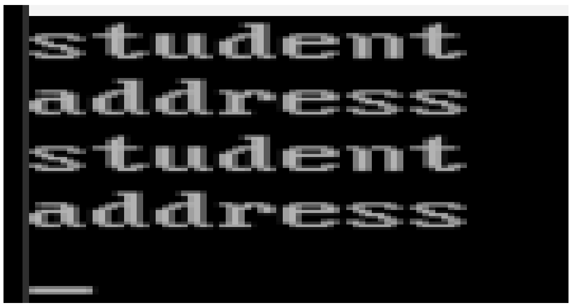
The rules for the virtual functions in C++ are as follows:

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have a virtual destructor but it cannot have a virtual constructor.

SAMPLE PROGRAM WITHOUT RUN - TIME POLYMORPHISM:

```
#include<iostream.h>
#include<conio.h>
class one
{
	public:
		void student()
		{
			cout<<"student"<<endl;
		}
		void address()
		{
			cout<<"address"<<endl;
		}
};
class two:public one
{
	public:
		void student()
		{
			cout<<"derived student"<<endl;
		}
		void address()
		{
			cout<<"derived address"<<endl;
		}
};
void main()
{
	clrscr();
	one a;
	two b;
	one *ptr;
	ptr=&a;
	ptr->student();
	ptr->address();
	ptr=&b;
	ptr->student();
	ptr->address();
	getch();
}
```

*OUTPUT :*

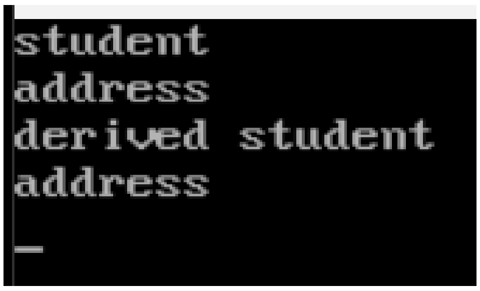


```
student
address
student
address
_
```

*SAMPLE PROGRAM WITH VIRTUAL FUNCTIONS :*

```
#include<iostream.h>
#include<conio.h>
class one
{
        public:
                virtual void student()
                {
                        cout<<"student"<<endl;
                }
                void address()
                {
                        cout<<"address"<<endl;
                }
};
class two:public one
{
        public:
                void student()
                {
                        cout<<"derived student"<<endl;
                }
                void address()
                {
                        cout<<"derived address"<<endl;
                }
};
void main()
{
        clrscr();
        one a;
        two b;
        one *ptr;
        ptr=&a;
        ptr->student();
        ptr->address();
        ptr=&b;
        ptr->student();
        ptr->address();
        getch();
}
```

*OUTPUT :*



```
student
address
derived student
address
_
```

**Limitations of Virtual Functions**

- Slower: The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.
- Difficult to Debug: In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

***PURE VIRTUAL FUNCTIONS :***

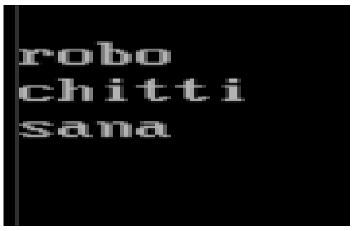
- A pure virtual function (or abstract function) in C++ is a virtual function for which we can have an implementation, But we must override that function in the derived class, otherwise, the derived class will also become an abstract class. A pure virtual function is declared by assigning 0 in the declaration..

*SAMPLE PROGRAM FOR PURE VIRTUAL FUNCTION :*

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

```
{
        public:
                virtual void robo()=0;
                virtual void chitti()=0;
                void sana()
                {
                        cout<<"sana"<<endl;
                }
};
class B:public A
{
        public:
                void robo()
                {
                        cout<<"robo"<<endl;
                }
                void chitti()
                {
                        cout<<"chitti"<<endl;
                }
};
void main()
{
        clrscr();
        B b;
        b.robo();
        b.chitti();
        b.sana();
        getch();
}
```

*OUTPUT :*



```
robo
chitti
sana
```

***LATE BINDING AND EARLY BINDING***

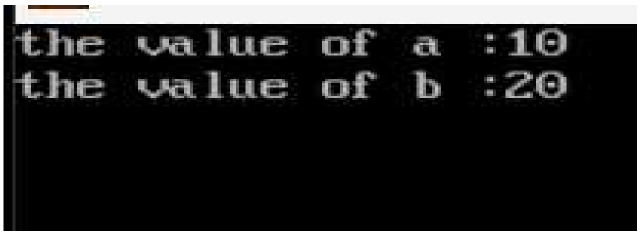
| ***Aspect*** | ***Early Binding (Static Binding)*** | ***Late Binding (Dynamic Binding)*** |
|---|---|---|
| *Timing of Binding* | *Occurs at compile-time.* | *Occurs at runtime.* |
| *Decision Time* | *Decisions are made before execution.* | *Decisions are made during program execution.* |
| *Mechanism* | *Function overloading, templates.* | *Virtual functions and polymorphism.* |
| *Implementation* | *Resolved at compile-time.* | *Resolved at runtime.* |
| *Flexibility* | *Less flexible, as decisions are fixed during compilation.* | *More flexible, as decisions can change at runtime based on the actual object type.* |
| *Examples* | *Function overloading, templates.* | *Virtual functions, polymorphism.* |
| *Code Example (Partial)* | *cpp Base obj; obj.display();* | *cpp Base* ptr = &obj; ptr->display();* |
| *Keyword (if applicable)* | *Not applicable.* | *virtual keyword is used with functions in the base class.* |
| *Common Use Case* | *Compile-time optimization, performance considerations.* | *Achieving runtime polymorphism, flexibility in design.* |

***THIS POINTER :***

In C++, the this pointer is a keyword that represents a pointer to the current instance of a class. It is a hidden parameter that is passed to all member function calls. The this pointer is used to access the members of the class within its own member functions, especially when there is a need to disambiguate between member variables and parameters with the same name.

```
#include<iostream.h>
#include<conio.h>
class A
{
        int a,b;
        public:
                A()
                {
                        a=0;
                        b=0;
                }
                void get(int a,int b)
                {
                        this->a=a;
                        this->b=b;
                }
                void show()
                {
                        cout<<"the value of a :"<<a<<endl;
                        cout<<"the value of b :"<<b<<endl;
                }
};
void main()
{
        A a;
        clrscr();
        a.get(10,20);
        a.show();
        getch();
}
```

*OUTPUT :*



```
the value of a :10
the value of b :20
```

## TEMPLATES

Templates in C++ are a powerful feature that allows you to write generic code. Templates can be used for functions, classes, and even entire programs

Templates in C++ offer several advantages, making them a powerful and flexible feature. Here are some key advantages:

Code Reusability:

> Templates allow you to write generic code that can work with multiple data types. This promotes code reusability since the same template can be used for different types without duplicating the code.

Type Safety:

> Templates maintain type safety by allowing you to write generic code that is still checked at compile time for type correctness. This helps catch errors early in the development process.

Flexibility:

> Templates provide a flexible way to create generic algorithms and data structures. This flexibility enables you to write code that adapts to different data types and structures seamlessly.

Performance:

> Template code is generated at compile time, which can result in more efficient code execution compared to runtime polymorphism. In situations where performance is crucial, templates offer a way to achieve high-performance generic solutions.

Standard Template Library (STL):

> The C++ Standard Template Library (STL) extensively uses templates. Containers like vectors, lists, and maps, as well as algorithms like sort and find, are implemented as templates. This allows for a consistent and generic approach to handling various data structures and algorithms.

Compile-Time Polymorphism:

Templates enable compile-time polymorphism, as opposed to runtime polymorphism offered by virtual functions and inheritance. This can lead to more efficient code execution since decisions are made at compile time.

Template Specialization:

C++ templates support specialization, allowing you to provide specific implementations for certain types. This can be useful when you need to customize behavior for specific data types while maintaining a generic solution for others.

Clearer Code:

Templates often result in more concise and readable code, as they allow you to express generic algorithms without the need for extensive type-specific code. This can lead to clearer and more maintainable code.

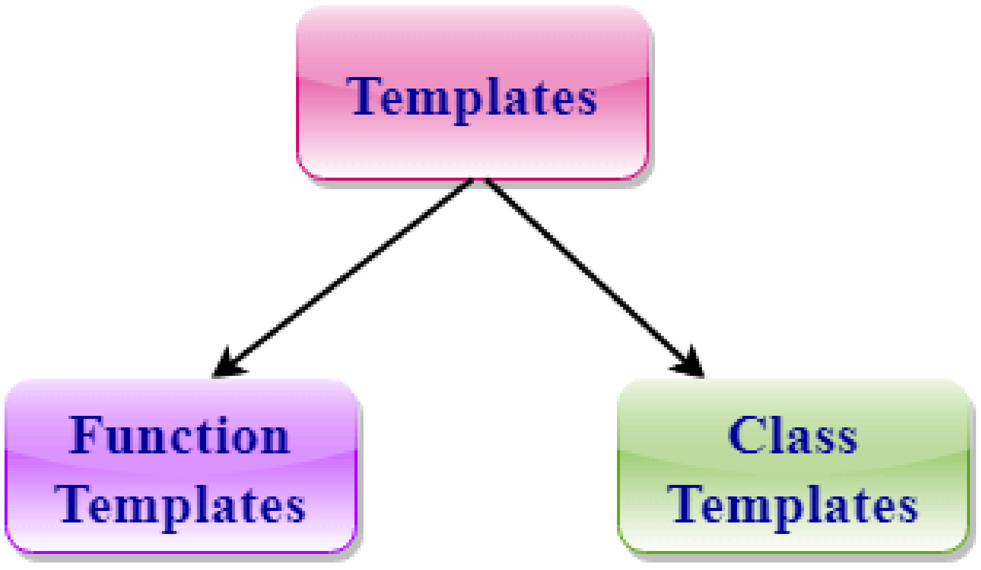
Generic Programming:

Templates support generic programming, a programming paradigm that focuses on writing algorithms and data structures in a way that they can operate on any data type. This approach leads to more versatile and adaptable code.

In summary, templates in C++ contribute to code reusability, type safety, flexibility, and performance, making them a valuable tool for generic programming and the creation of versatile software components.

Templates are of two types :

1. FUNCTION TEMPLATES
2. CLASS TEMPLATES



FUNCTION TEMPLATES VS CLASS TEMPLATES :

Certainly, here's a tabular representation summarizing key aspects of function templates and class templates in C++:

| Aspect | Function Templates | Class Templates |
|---|---|---|
| Definition Syntax | template <typename T> returnType functionName(parameters); | template <typename T1, typename T2> class ClassName { /* ... */ }; |
| Generic Types | Single generic type (e.g., T) | Multiple generic types (e.g., T1, T2) |
| Use in Functions | Used for generic functions | Used for generic classes |
| Example Function | cpp int max(T a, T b) { return (a > b) ? a : b; } | Not applicable (class template example provided earlier) |

|  |  |  |
|---|---|---|
| Example Usage | cpp int resultInt = max(5, 10); double resultDouble = max(3.14, 2.71); | cpp Pair<int, std::string> myPair(42, "Hello"); |
| Type Safety | Ensures type safety at compile time | Ensures type safety at compile time |
| Template Specialization | Can be specialized for specific types | Can be specialized for specific types |
| Code Reusability | Enables reuse with different data types | Enables reuse with different data types |
| Compile-time Polymorphism | Offers compile-time polymorphism | Offers compile-time polymorphism |
| STL Usage | Used in algorithms like std::max and std::sort | Used in containers like std::vector and algorithms like std::find |
| Flexibility | Provides flexibility in handling different data types | Provides flexibility in handling different data types |

These templates play a crucial role in generic programming, allowing developers to write versatile and efficient code that can adapt to various data types and structures.

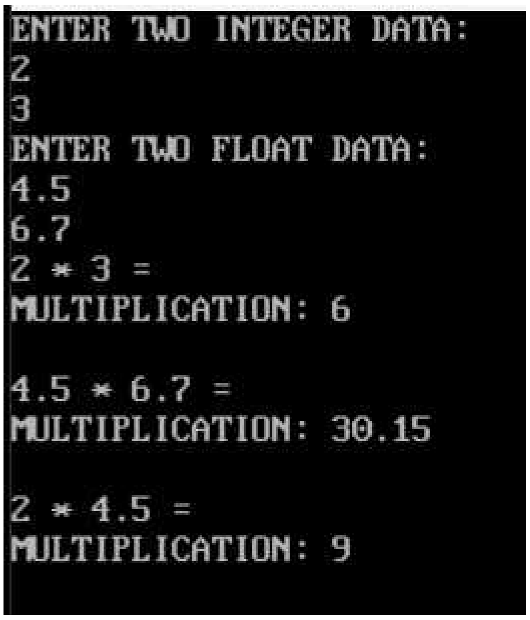
SAMPLE CODES :

A) FUNCTION TEMPLATES

```
#include<iostream.h>
#include<conio.h>
template<class T, class U>
void multiply(T a,U b)
{
cout<<"MULTIPLICATION: "<<a*b<<endl;
}
```

```
int main()
{
int a,b;
float x,y;
clrscr();
cout<<"ENTER TWO INTEGER DATA: "<<endl;
cin>>a>>b;
cout<<"ENTER TWO FLOAT DATA: "<<endl;
cin>>x>>y;
multiply(a,b);
multiply(x,y);
multiply(a,x);
getch();
return 0;
}
```

OUTPUT :



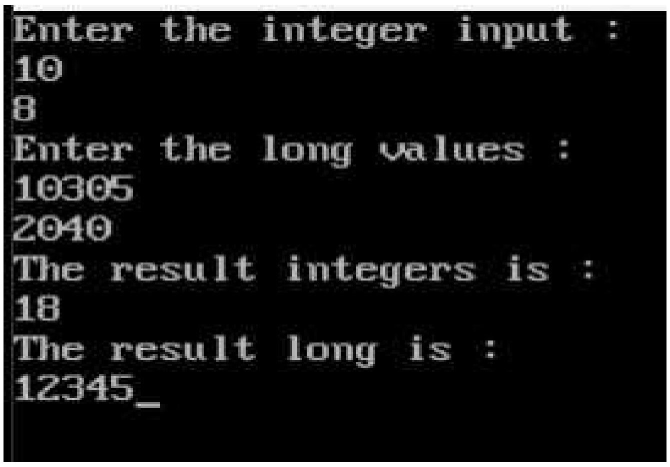
```
ENTER TWO INTEGER DATA:
2
3
ENTER TWO FLOAT DATA:
4.5
6.7
2 * 3 =
MULTIPLICATION: 6

4.5 * 6.7 =
MULTIPLICATION: 30.15

2 * 4.5 =
MULTIPLICATION: 9
```

## B) CLASS TEMPLATE

```
 #include<iostream.h>
#include<conio.h>
template<class T>
class addition
{
        public:
                T add(T,T);
};
```

```
template<class T>
T addition<T>::add(T n1,T n2)
{
        T result;
        result=n1+n2;
        return result;
}
int main()
{
        int a,b;
        long c,d;
        float result;
        addition<int>x;
        addition<long>y;
        clrscr();
        cout<<"Enter the integer input :"<<endl;
        cin>>a>>b;
        cout<<"Enter the long values :"<<endl;
        cin>>c>>d;
        cout<<"The result integers is :"<<endl;
        result=x.add(a,b);
        cout<<result;
        cout<<"\nthe result long is :"<<endl;
        result=y.add(c,d);
        cout<<result;
        return 0;
}
```

OUTPUT :



```
Enter the integer input :
10
8
Enter the long values :
10305
2040
The result integers is :
18
The result long is :
12345_
```

## FILE HANDLING

File :

File Is A Collection Of Records Where Each Record Consists Of Number Of Items .

These Files Can Be Arrange In Three Ways :

- ❑ Ascending /Descending Order
- ❑ Alphabetical Order
- ❑ Chronological Order
    1. Order Of Occurances Like Accoding To Date Or Size Or Events

Stream :

In C++ , A Stream Is A Data Flow From Keyboard To Monitor .

For Input And Output They Are Two Different Streams.

- ❑ Cin – Standard Input Stream
- ❑ Cout – Standard Output Stream
- ❑ Cerr – Standard Error Stream
    - ♦ Ifstream :
        - ● It Is The Input File Stream Class .
        - ● It Can Inherit The Folloing Functions .
            - → Open()
            - → Get()
            - → Getline()
            - → Read() Etc ...
    - ♦ Ofstream :
        - ● It Is The Output File Stream Class .
            - → Open()
            - → Put()
            - → Write()
    - ♦ Fstream :
        - ● It Supports Files For Input And Output .

**Functions :**

| | |
|---|---|
| Open | Opens The File . |
| Close | Closes The File . |
| Close All | It Closes All The Opened Streams |
| Seekg | Sets Current “Get” Position |
| Seekp | Sets Current “Put“ Position |
| Tellg | Returns “Get” |
| Tellp | Returns “Put” |

Opening And Closing A File :

We Can Open A File In 2 Ways :

- Using Constructors Of The Stream Class .
- By Using Open() Function .

Opening The File Using Open() Function :

- Declare A New File In Ofstream Or Output Stream .
- Assign File Stream As “Test.Dat” That Means

New File=”Test.Dat”

**Advantages:**

More Than One File Can Be Opened At A Time In Program .

Storing Files :

- If The Information Is Very Important Then We Must Try To Save It On Hard Disk So That It Can Be Reused .
- Normally , Contents Of Objects Are Lost As Soon As Object Goes Out Of Scope Or Execution Is Over .
- The Objects Which Remeber Their Data Are Called Persistent Objects .

Detecting End Of File :

- In Some Cases, We Don’t Know Where The File Is Going To End .
- A Simple Method Know To Know Is By Testing The Strream In While Loop

```
While(<Stream>)
{
        //
}
```

- The <Stream> Returns 0 Then As Soon As End Of The File Is Detected .

**Graphics Handling :**

Introduction :

- Graphics In C++ Is Defined To Create A Graphic Model By Creating Different Shapes And Adding Colours To It.
- It Can Be Done In C++ Console By Importing Graphics.H Library To Gcc Compiler.
- We Can Draw Circle, Line, Ellipse And Other Geometric Shapes Too.
- The Application Of Oops Is A Primary Technique To Be Used Here.
- C++ Does Not Have Any Inbuilt Funtion To Perform Drawing As They Have Low – Level Programs To Use; Instead ,We Can Use Api To Do Graphics.

**Formal  Syntax :**

The Formal Syntax Is Given As:

```
#include<graphics.h>

{

Initgraph();

}
```

ATTRIBUTES IN GRAPHICS :

- setcolor(color)
- setbkcolor(color)
- setlinestyle(style,pattern,thickness)

INITGRAPH :

- INITGRAPH IS USED TO INITIALIZE THE SYSTEM GRAPHICS BY LOADING A GRAPHICS DRIVER FROM DISK AND THEREBY PUTTING THE SYSTEM INTO GRAPHICS MODE .
- INITGRAPH HAS 3 PARAMETERS :

1)GRAPHIC DETECT

2)GRAPHIC MODE

3)BGI  FILE :

I)BGI STANDS FOR BORLAND GRAPHICS INTERFACE ,IT IS A GRAPHIC LIBRARY.

II)THIS LIBRARY LOADS GRAPHIC DRIVER AND VECTOR FONTS(*.CHR)

CLOSEGRAPH:

CLOSEGRAPH DELLOCATES THE MEMORY ALLOCATED BY THE SYSTEM GRAPHICS AND THEN RESTORES THE SCREEN TO THE MODEIT WAS BEFORE CALLING INITGRAPH.

COLOUR FUNCTIONS :

- IN C++ GRAPHICS THERE ARE 16 COLORS DECLARED
- WE USE THESE 16 COLOURS TO CHANGE BACKGROUND COLOUR, FONT COLOUR, CHANGE COLOUR OF SHAPES, FILL COLOUR OF SHAPES, CHANGE TEXT COLOUR .

TEXTCOLOR :

TEXTCOLOR IS USED TO SET THE TEXT COLOUR.

textcolor(green);

cout<<"TEXT IN GREEN "<<endl;

SETBKCOLOR :

SETBKCOLOR IS USED TO SET THE BACKGROUND COLOUR BY

SPECIFYING THE COLOUR NAME OR NAME THE NUMBER.

setbkcolor(blue);

rectangle(100,100,200,200);

SETCOLOR :

SETCOLOR IS USED TO SET THE TEXT COLOR OR SET THE OUTLINE COLOUR OF VARIOUS SHAPES .

setcolor(green);  (or) setcolor(3);

rectangle(100,100,200,200);

| COLOUR NAME | COLOUR VALUE |
|---|---|
| BLACK | 0 |
| BLUE | 1 |
| GREEN | 2 |
| CYAN | 3 |
| RED | 4 |
| MAGENTA | 5 |
| BROWN | 6 |
| LIGHT GRAY | 7 |
| DARK GRAY | 8 |
| LIGHT BLUE | 9 |
| LIGHT GREEN | 10 |
| LIGHT CYAN | 11 |
| LIGHT RED | 12 |
| LIGHT MAGENTA | 13 |
| YELLOW | 14 |
| WHITE | 15 |

OTHER FUNCTIONS :

- ☺ *BLINK : IT HELPS TO BLINK THE CHARACTER ON THE CONSOLE SCREEN WINDOW.*
- ☺ *GOTOXY : IT HELPS TO MOVE THE CURSOR TO ANY POSITION ON THE SCREEN.*
- ☺ *DELAY : IT SUSPENDS A FEW SECTIONS .*

For example , For moving a car;it waits for a while .

- ☺ GETMAXX : RETURNS TO MAXIMUM OF X COORDINATES .
- ☺ GETMAXY : RETURNS TO MAXIMUM OF Y COORDINATES .

CO-ORDINATES :

THIS SPECIFIES HOW POINTS ARE PLACED IN A WINDOW ; THE INITIAL POINT OF THE SCREENING POINT IS ASSUMED AS (0,0).

THIS CO-ORDINATED SYSTEM DEPICTS HOW AND WHERE TO PERFORM A DRAW OPTION IS SPECIFIED .

- ❑ THE GRAPHICS SCREEN HAS 640*480 PIXELS.

SHAPES :

- ★ CLEAR() - IT RETURNS THE CURSOR POSITION TO (0,0).
- ★ CIRCLE(x,y,radius) - IT CREATES A CIRCLE WITH A GIVEN RADIUS .
  - IT CAN BE DRAWN WITH THREE PARAMETERS .
  - x,y,radius ARE THREE PARAMETERS USED .
  - x,y ARE THE POINT OF CO-ORDINATES OF CENTRE .
- ★ LINE(x,y) - IT CREATES A LINE WITH STARTING AND ENDING POINTS .
  - IT CAN BE DRAWN WITH THREE PARAMETERS .
  - x,y,radius ARE THREE PARAMETERS USED .
  - (x1,y1) and (x2,y2) ARE THE POINT OF CO-ORDINATES OF EITHER ENDS .
- ★ RECTANGLE(x1,x2,y1,y2) :IT IS USED TO CREATE A RECTANGLE.
  - IT CAN BE DRAWN WITH THREE PARAMETERS .
  - x,y,radius ARE FOUR PARAMETERS USED .
  - x1,x2,y1,y2 ARE THE POINT OF CO-ORDINATES OF EITHER ENDS .
    - → X1(LEFT)
    - → Y1(TOP)
    - → X2(RIGHT)
    - → Y2(BOTTOM )

NAMESPACE::

In C++, namespaces are used to organize code by grouping related elements together, preventing naming conflicts. They are defined using the namespace keyword and accessed using the scope resolution operator ::. Namespaces are beneficial for maintaining code clarity and avoiding issues in large projects or when integrating different libraries. The using namespace directive can simplify code, but it should be used cautiously to avoid unintended conflicts.

```
#include <iostream>
// First namespace
namespace Math
 {
   int add(int a, int b)
          {
                   return a + b;
          }

   int subtract(int a, int b)
          {
                  return a - b;
```

```
            }
}
// Second namespace
namespace Utility
 {
   void displayMessage(const char* message)
          {
                  std::cout << message << std::endl;
          }
}
int main()
{
   // Using functions from the Math namespace
   int sum = Math::add(10, 5);
   int difference = Math::subtract(10, 5);

   std::cout << "Sum: " << sum << std::endl;
   std::cout << "Difference: " << difference << std::endl;

   // Using function from the Utility namespace
   Utility::displayMessage("Hello, namespaces!");

   return 0;
}
```

## ERROR HANDLING AND EXCEPTION HANDLING ::

| Aspect | Error Handling | Exception Handling (C++) |
|---|---|---|
| Definition | Broad concept dealing with both expected and unexpected errors. | Specific mechanism for dealing with exceptional situations using try, catch, and throw. |
| Language Support | Present in most programming languages. | C++ has explicit support for exception handling. |

| | | |
|---|---|---|
| Handling Method | - Return values indicating success/failure.<br> - Error codes for specific error conditions.<br> - Global variables or flags.<br> - Logging. | - try block contains code that might throw an exception.<br> - throw statement raises an exception.<br> - catch block handles the exception.<br> - Standard exception classes (e.g., std::runtime_error). |
| Structured Approach | Less structured, error checks may be scattered throughout the code. | More structured, separates error-handling code from main logic. |
| Example (C++) | cpp int result = divide(10, 0); if (result < 0) { // handle error } | cpp try { int result = divide(10, 0); } catch (std::runtime_error& e) { // handle exception } |
| Use Cases | - General error situations where explicit handling is feasible.<br> - Commonly used in many languages. | - Handling exceptional situations that are not easily predictable.<br> - Provides a clean separation of normal and exceptional code paths.<br> - Used in C++ for robust error management. |

This tabular representation summarizes the key aspects of both error handling and exception handling in programming, with a focus on C++ for exception handling.

## EXCEPTION HANDLING SAMPLE CODE □:

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

float Division(float a, float b)
{
    if (b == 0)
    {
        throw runtime_error("Math error: Attempted to divide by Zero\n");
    }
    return (a / b);
}

int main()
{
    float a, b, result;
    a =12.5;
    b= 0;
    try
    {
        result = Division(a, b);
        cout << "The quotient is "<< result << endl;
    }
    catch (runtime_error& e)
    {
            cout << "Exception occurred" << endl << e.what();
    }

} // end main
```