LNCT GROUP OF COLLEGES
"WORKING TOWARDS BEING THE BEST"

UNIT 3

Basic Computer Engineering (BT-205)

SCHEME

Rajiv Gandhi Proudyogiki Vishwavidyalaya, Bhopal

New Scheme of Examination as per AICTE Flexible Curricula

Bachelor of Technology (B.Tech.)

II Semester (Group A)

W.E.F. JULY 2018

GROUP A: (CS, IT, EE, EX, EI, FT, AT, MI, BT, & BM)

S.No.	Subject Code	Category	Subject Name	Maximum Marks Allotted					Total Marks	Contact Hours per week			Total Credits
				Theory Slot			Practical Slot						
				End Sem.	Mid Sem Exam.	Quiz/ Assignment	End Sem.	Lab work & Sessional		L	T	P	
1.	BT201	BSC-3	Engineering Physics	70	20	10	30	20	150	2	1	2	4
2.	BT202	BSC-4	Mathematics-II	70	20	10	-	-	100	3	1	-	4
3.	BT203	ESC-4	Basic Mechanical Engineering	70	20	10	30	20	150	3	-	2	4
4.	BT204	ESC-5	Basic Civil Engineering & Mechanics	70	20	10	30	20	150	3	-	2	4
5.	BT205	ESC-6	Basic Computer Engineering	70	20	10	30	20	150	3	-	2	4
6.	BT206	HSMC-2	Language Lab & Seminars	-	-	-	30	20	50	-	-	2	1
7.	BT107	DLC-1	Internship-I (60 Hrs Duration) at the Institute level	To be completed during first/second semester. Its evaluation/credit to be added in third semester.									
			Total	350	100	50	150	100	750	14	2	10	21

1 Hr Lecture	1 Hr Tutorial	2 Hr Practical
1 Credit	1 Credit	1 Credit

SYLLABUS

RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL
New Scheme Based On AICTE Flexible Curricula
B. Tech. First Year
Branch- Common to All Disciplines

BT205	Basic Computer Engineering	3L-0T-2P	4 Credits

Course Contents:

UNIT I

Computer: Definition, Classification, Organization i.e. CPU, register, Bus architecture, Instruction set, Memory & Storage Systems, I/O Devices, and System & Application Software. Computer Application in e-Business, Bio-Informatics, health Care, Remote Sensing & GIS, Meteorology and Climatology, Computer Gaming, Multimedia and Animation etc.

Operating System: Definition, Function, Types, Management of File, Process & Memory. Introdcution to MS word, MS powerpoint, MS Excel

UNIT II

Introduction to Algorithms, Complexities and Flowchart, Introduction to Programming, Categories of Programming Languages, Program Design, Programming Paradigms, Characteristics or Concepts of OOP, Procedure Oriented Programming VS object oriented Programming. Introduction to C++: Character Set, Tokens, Precedence and Associativity, Program Structure, Data Types, Variables, Operators, Expressions, Statements and control structures, I/O operations, Array, Functions,

UNIT III

Object & Classes, Scope Resolution Operator, Constructors & Destructors, Friend Functions, Inheritance, Polymorphism, Overloading Functions & Operators, Types of Inheritance, Virtual functions. Introduction to Data Structures.

UNIT IV

Computer Networking: Introduction, Goals, ISO-OSI Model, Functions of Different Layers. Internetworking Concepts, Devices, TCP/IP Model. Introduction to Internet, World Wide Web, E-commerce

Computer Security Basics: Introduction to viruses, worms, malware, Trojans, Spyware and Anti-Spyware Software, Different types of attacks like Money Laundering, Information Theft, Cyber Pornography, Email spoofing, Denial of Service (DoS), Cyber Stalking, ,Logic bombs, Hacking Spamming, Cyber Defamation , pharming Security measures Firewall, Computer Ethics & Good Practices, Introduction of Cyber Laws about Internet Fraud, Good Computer Security Habits,

UNIT V

Data base Management System: Introduction, File oriented approach and Database approach, Data Models, Architecture of Database System, Data independence, Data dictionary, DBA, Primary Key, Data definition language and Manipulation Languages.

Cloud computing: definition, cloud infrastructure, cloud segments or service delivery models (IaaS, PaaS and SaaS), cloud deployment models/ types of cloud (public, private, community and hybrid clouds), Pros and Cons of cloud computing

List of Experiment

01. Study and practice of Internal & External DOS commands.
02. Study and practice of Basic linux Commands – ls, cp, mv, rm, chmod, kill, ps etc.
03. Study and Practice of MS windows – Folder related operations, My-Computer, window explorer, Control Panel,
04. Creation and editing of Text files using MS- word.
05. Creation and operating of spreadsheet using MS-Excel.
06. Creation and editing power-point slides using MS- power point
07. Creation and manipulation of database table using SQL in MS-Access.
08. WAP to illustrate Arithmetic expressions
09. WAP to illustrate Arrays.
10. WAP to illustrate functions.
11. WAP to illustrate constructor & Destructor
12. WAP to illustrate Object and classes.

Classes

- A class is a user-defined data type.
- class holds data members and member functions.
- It is a blueprint for objects in Object-Oriented Programming (OOP).
- Example: Class of birds — all birds can fly, have wings and beaks.
- A class works as “template" for creating objects.
- Central idea in oops is Placing data & functions in a single entity.
- Each class is collection of data & functions that manipulate the data.
- When we define a class, we only define specifications for the object.
- No storage or memory is allocated while defining a class.
- To use the data and access functions defined in a class, we must create objects.

Properties of a Classes

- Class name should start with an uppercase letter (convention).
- Contains data members and member functions.
- Access to members is controlled by access specifiers.
- Member functions can be defined inside or outside the class.
- Similar to C structures, but defaults to private access.
- Supports OOP features: Inheritance, Encapsulation, Abstraction.
- Objects hold separate copies of data members.

Classes

```
#include <iostream>
using namespace std;
class rectangle
 {                    // Class definition
            public:
            int len,br; // Data member
            void get_data( );
             void area( );
             void print_data( );
};
```

Syntax:

```
class Class_Name
 {                    // Class definition
private:
                    // Data member
public:
                    // Methods
};
```

Classes

```
#include <iostream>
using namespace std;
class Fruit {                    // Class definition
public:
      string color =“red”; // Data member
       void show()
       {              // Member function
             cout << "Fruit color is : " << color << endl;
       }
};
int main() {
         Fruit apple;          // Object declared
         apple.show();
         return 0;
}
```

Inside Class Definition

```
#include <iostream>
using namespace std;
class Car {
public:
    void Brand() {
                        // class Function definition
                          cout << "Car Brand: Tesla" << endl;
                    }
};
int main() {
    Car c1;
    c1.Brand();  // Calling member function
    return 0;
}
```

Outside Class Definition

```
#include <iostream>
using namespace std;
class Car {
public:
                void Brand();  // Function declaration(prototype) inside class
              };
int main() {
                  Car c2;
                  c2.Brand();  // Calling member function
                  return 0;
                }
// Function definition outside class
void Car::Brand() {
                                  cout << "Car Brand: BMW" << endl;
                                }
```

Object

- An object used to represent real-world concepts and entities.
- Objects are instances of a class.
- Memory is allocated when an object is created.
- Object can be created using the class name.
- Object interacts with the help of methods defined within class.
- For example, the human type student is a class, while a particular student named Ram is an object of the student class.
- an object is a fundamental concept in Object-Oriented Programming.
- It represents a concrete instance of a class.
- Objects are the actual entities that are created as an instance of a class. There can be as many objects of a class as desired.
- Ex. int a; //an instance of type integer
- student ram; //an instance of type student

Characteristics of an Object

1. Instance of a Class:

- A class acts as a blueprint or a template.
- It defines the structure and behavior (data members and member functions) for a specific type of entity.
- An object is a tangible realization of that blueprint.
- it's a specific entity created based on the class definition.

2. Encapsulation of Data and Behavior:

- An object bundles together data (attributes or member variables) and the functions (methods or member functions).
- It operate on that data into a single unit. This principle is known as encapsulation.

Characteristics of an Object

3. State and Behavior:

- **State:** The state of an object is defined by the values of its data members at a particular point in time.
- **Behavior:** The behavior of an object is determined by the actions it can perform through its member functions.

4. Memory Allocation:

- When a class is defined, no memory is allocated.
- Memory is only allocated when an object of that class is created.
- Each object of a class will have its own separate memory space for its data members.

Creating an Object

- Once the class is defined, we can create its object in the same way we declare the variables of any other inbuilt data type.

ClassName objectName;

- **Member Access :** Members of the class can be accessed inside the class itself simply by using their assigned name.
- To access them outside, the (.) dot operator is used with the object of the class.
- **obj.member1** // For data members
- **obj.member2(..)** // For functions

Creating an Object

```
#include <iostream>
using namespace std;
class Box {
public:
    double length, height;
};
int main() {
    Box Box1, Box2;
    // Assign values and
   compute volume
}
```

Creating an Object

```
#include <iostream>
using namespace std;
class Example
{
private:
int a,b,c;    //data member
public:
int add( int a, int b)
 {
   c=a+b;
 }
void print()
 {
   cout<<“sum is : “<<c<<“\n”;
 }
};
int main()
{
Example obj;
                    // object created
Obj.add(10,20);
                    //function call
Obj.print( );

return 0;
}
```

Creating an Object

```
#include <iostream>
using namespace std;
class Car
{
// Class definition
public:
string brand;
};
int main()
{
Car myCar;
                // Object create
  myCar.brand = “BMW";
Car myCar2;
                // Object create
myCar.brand = "Toyota";
                // Assign value
cout << "Car Brand: " <<myCar.brand;
                // Access object data
return 0;
}
```

LNCT GROUP OF COLLEGES
"WORKING TOWARDS BEING THE BEST"

Object Vs Classes

Class	Object
For a single class there can be any number of objects. Ex. For River class , ganga Yamuna, Narmada can be objects.	There are many objects that can be created from one class. These objects make use of method and attributes defined by belonging class.
Scope of class is persistent throughout the program.	Objects can be created & destroyed as per the requirements.
Class can not be initialized with some property value.	We can assign some property values to the objects.
A class has unique name	Various objects having different names can be created for the same class.

LNCT GROUP OF COLLEGES
"WORKING TOWARDS BEING THE BEST"

Object Vs Classes

Class	Object
A blueprint or template for creating objects.	An instance of a class with actual values.
No memory is allocated for a class until an object is created.	Memory is allocated when an object is created.
Conceptual entity describing structure and behaviour.	A real-world entity created from the class.
Defines properties and functions common to all objects of that type.	Stores specific data and manipulates it using class functions.
Represents a general concept or type.	Represents a specific instance of the class.

LNCT GROUP OF COLLEGES
"WORKING TOWARDS BEING THE BEST"

Structure Vs Classes

Structure	Class
By default members of structure are public.	By default members of class are private.
Structure can not be inherited.	Class can be inherited.
Structure do not require constructors.	Classes require constructors for initializing the objects.
Structure contains only data members.	Class contains the data as well as the function members.

Access Specifiers

- Public: Accessible from anywhere.
- Private: Accessible only within the class (default).
- Protected: Accessible within the class and derived classes.
- Example:
- class PublicAccess { public: int x; void display(); };
- class PrivateAccess { private: int x; void display(); };
- class ProtectedAccess { protected: int x; void display(); };

Scope Resolution Operator

- The scope resolution operator in C++ is denoted by two colon symbols (::).
- It is a powerful operator used to specify the scope to which an identifier (variable, function, class, or namespace) belongs.
- This is particularly useful in situations where there might be name conflicts or when you need to explicitly refer to an entity within a specific scope.
- Scope resolution operator (::) is used to access the identifiers such as variable names and function names defined inside some other scope in the current scope.
- Uses:
- Access global variables when local variables have the same name.
- Define functions outside a class.
- Access static members of a class.
- Resolve ambiguity in multiple inheritance.

Scope Resolution Operator

- **Accessing Global Variables:** When a local variable has the same name as a global variable, the local variable takes precedence within its scope.
- To access the global variable explicitly, the scope resolution operator can be used.

```
int x = 10; // Global variable
int main() {
int x = 20; // Local variable
cout << "Local x: " << x ;
cout << "Global x: " << ::x ;
return 0;    }
```

Scope Resolution Operator

- **Defining Member Functions Outside a Class:** Member functions of a class can be defined outside the class definition using the scope resolution operator to associate the function with its respective class.

```
class Fruit {
public:
   void Print_Color();
};

void Fruit::Print_Color() {
   // Function definition
}
```

Scope Resolution Operator

- **Accessing Static Members of a Class:** Static members (variables or functions) belong to the class. They are accessed using the class name followed by the scope resolution operator and the member name.

```
class MyClass {
public:
    static int count;
};
int MyClass::count = 0; // Definition of static member

int main() {
    cout << MyClass::count;
    return 0;
}
```

Scope Resolution Operator

- **Referring to Members of a Namespace:** To access members (variables, functions, classes, etc.) declared within a namespace, the namespace name is used, followed by the scope resolution operator and the member name.

```
namespace MyNamespace
{
    int a = 5;
}
int main() {
    cout << MyNamespace::a;
    return 0;
}
```

LNCT GROUP OF COLLEGES
"WORKING TOWARDS BEING THE BEST"

Constructors

- A constructor is a special method(special class members).
- It is automatically called when an object of a class is created.
- Constructors are called by the compiler every time an object of that class is instantiated.
- Constructors share the same name as the class.
- It can be defined inside or outside the class definition.
- Special member function with the same name as the class.
- No return type, not even void.
- Automatically invoked when an object is created.
- Types of constructors :

1. **Default Constructor: No parameters.**
2. **Parameterized Constructor: Takes parameters.**
3. **Copy Constructor: Copies one object to another.**

Characteristics of a Constructor

- Declared in public section.
- No return type because Constructors do not return values.
- Cannot be inherited or virtual.
- Cannot have their address taken.
- If we do not specify a constructor, Compiler generates a default constructor for us (expects no parameters and has an empty body).
- The name of the constructor is the same as its class name.
- A constructor gets called automatically when we create the object of the class.
- Multiple constructors can be declared in a single class.
- In case of multiple constructors, the one with matching function signature will be called.

Constructors

```
class Fruit
{
    public:

   Fruit()
     {
        cout << “Constructor called";
      }
   };

   int main() {
     Fruit Orange;
// Create an object of fruit Class
//(this will call the constructor)
     return 0;
   }
```

Output:
Constructor called

Ex.2

```
class Line {
public:
    Line(); // constructor
};
Line::Line() {
    cout << "Object created";
}
```

Default Constructors

```
#include <iostream>
using namespace std;
// Class with no explicity defined
   constructors
class Fruit
{
public:
};
int main() {
          // Creating object
    Fruit apple;
    return 0;
}
```

No constructor defined in class means default constructor called.

Default Constructors

```
#include <iostream>
using namespace std;
class Fruit{
public:
    string color;
    // Default Constructor
    Fruit() {
        color = “Red";
        cout << "Default Constructor
   called:"<<" Fruit Color is " << color;
    }
    };
int main() {
    Fruit apple;     // Default Constructor
    return 0;
    }
```

Output:
Default Constructor called: Fruit Color is Red

1. A default constructor is automatically generated by the compiler if the programmer does not define one.
2. This constructor doesn't take any argument as it is parameter less and initializes object members using default values.
3. It is also called a zero-argument constructor.

Parameterized Constructors

```
#include <iostream>
using namespace std;
class ABC {
public:
    int val;

    // Parameterized Constructor
    ABC(int x) {
        val = x;
    }
};

int main() {

    // Creating object with a parameter
    ABC a(10);
    cout << a.val;
    return 0;
}
```

Output:
10

Parameterized Constructors

```
#include <iostream>
using namespace std;
class Fruit {
public:
    string color;
    //  Parameterized Constructor
    Fruit(string b) {
        color = b;
        cout << "Parameterized
    Constructor: fruit color is " << color
    << endl;
    }
};
int main() {
    Fruit apple("Green");
    // Parameterized Constructor
    return 0;
}
```

Output:
Parameterized Constructor: fruit color is Green

1. Parameterized constructor allow us to pass arguments to constructors.
2. these arguments help initialize an object's members.
3. To create a parameterized constructor, simply add parameters to it the way you would to any other function.
4. When you define the constructor's body, use the parameters to initialize the object's members.

Copy Constructors

```
#include <iostream>
using namespace std;
class Fruit{
public:
   string color;
Fruit(string b) {          //  Parameterized Constructor
       color = b;
       cout << "Parameterized Constructor: color is " <<
   color<< endl;
   }
Fruit(Fruit &f) {          //  Copy Constructor
       color = f.color;
       cout << "Copy Constructor data for new fruit now new
   fruit color is " << color << endl;
   }
};
int main() {
   Fruit apple("Yellow");       // Parameterized Constructor
   Fruit mango = apple;          // Copy Constructor
      return 0;
}
```

Output:

Parameterized Constructor: color is Yellow

Copy Constructor data for new fruit now new fruit color is Yellow

Copy Constructors

```
#include <iostream>
using namespace std;
class A {
public:
    int val;
        // Parameterized constructor
    A(int x) {
                    val = x;
                }
    // Copy constructor
    A(A &a) {
                     val = a.val;     }     };
int main() {
    A a(20);
                // Creating another object from a
    A newobj(a);
    cout << newobj.val;
    return 0;
}
```

Output:
20

Destructors

- Destructors are called by the compiler when the **scope** of an object **ends**.
- They deallocate memory earlier used by the object of the class to avoid any **memory leaks**.
- Destructors have the same name as the class but **Prefixed with ~.**
- Destructor is an instance member function that is **invoked automatically** whenever an object is going to be destroyed.
- destructor is the **last function** that is going to be called before an object is destroyed.
- Special function called when an object goes out of scope.
- **No arguments**.
- Syntax:

```
~className(){     // Body of destructor   }
```

Destructors

```
class A {
public:
   ~A(); // destructor
};
```

Destructors

```
#include <iostream>
using namespace std;
class File {
public:
    File() {
        cout << "File Opened!" << endl;
            }

    ~File() {
        cout << "File Closed!" << endl;
             }
};
int main() {
    File f1;  // Constructor
    return 0;  // Destructor
}
```

Destructors

```
class A {
  public:
  A()
  {
        cout << "Constructor Called “; }
  ~A() {
        cout << "Destructor Called”;
          }   };
  int main() {
     A obj;
     return 0;
  }
```

Characteristics of a Destructor

- Destructor has the same name as their class name
- preceded by a tilde (~) symbol.
- Only one destructor is defined in a class.
- destructor cannot be overloaded.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

Friend Functions

- Friend function is a **non-member** function.
- It is Defined **outside** the class.
- It is Used to access private and protected members.
- It is Declared with the friend keyword.
- it is declared within the class definition using the friend keyword.
- it is not considered a member of that class.
- it is **not** invoked using the member-selection operators (. or ->).
- Friend function allow to access the private and protected members of another class.
- Syntax:

```
class A {
   friend void fun();
};
```

Friend Functions

- Example:

```
class class_name {
                        private:
                        //data
                        friend return_type friend_fun_name( );   //declare
                      };
return_type friend_fun_name(){
                                        //Object create
                                        //private data access here
                                    }

int main() {
                friend_fun_name();    //call
                return 0;
              }
```

Why we use Friend Function?

```
class ABC {
    private:
        int a;
}
int main() {
    ABC obj;
    // Error! Cannot access private members from here.
    obj.a = 5;
}
```

- friend functions that break this rule and allow us to access private member functions from outside the class.

Friend Functions

```
#include <iostream>   using namespace std;
class A {
            private:
            int a;
            friend void fun();
        };
void fun(){
              A obj;
              obj.a=10;
              cout<<"friend function update private  data:"<< obj.a;
             }

int main() {
              fun();
              return 0;
             }
```

Characteristics Friend Functions

1. Access to private and protected members
2. Declared inside the class
3. Defined outside the class
4. Not a member function
5. Can be member function of another class.

- **Friend Functions Use**
- **Operator overloading:** For overloading binary operators.
- **Utility functions:** For implement global utility functions.
- **Testing:** friend functions used to test private state of a class.

Advantage Friend Functions

- **No Inheritance** :- A friend function used to access members without the need of inheriting the class.
- **Bridge b/w classes:-** The friend function acts as a bridge between two classes by accessing their private data.
- **Versatile:-** It can be used to increase the versatility of overloaded operators.
- **Declaration:-** It can be declared either in the public or private or protected part of the class.

Demerits of Friend Functions

- **Violate data hiding**:- Friend functions have access to private members of a class from outside the class which violates the law of data hiding.
- **No Runtime Polymorphism:-** Friend functions cannot do any run-time polymorphism in their members.
- **Less Effective Encapsulation:-**Declaring too many functions or external classes as friends with access to a class's private or protected data reduces the effectiveness of encapsulation. This compromises one or more of the core principles of object-oriented programming.
- **Non-inheritable**:- Friendship is not inherited. In layman's terms, if a base class has a friend function, then the function doesn't become a friend of the derived class(es).

Friend Functions

```
class A {
        private:
        int a=10;
        friend int fun();
      };

int fun(){
        A obj;
        return obj.a ;
      }
 int main() {
            cout<<"friend function access private variable a:";
            cout<<fun();
            return 0;
          }
```

- friend functions that break this rule and allow us to access private member functions from outside the class.

Friend Functions

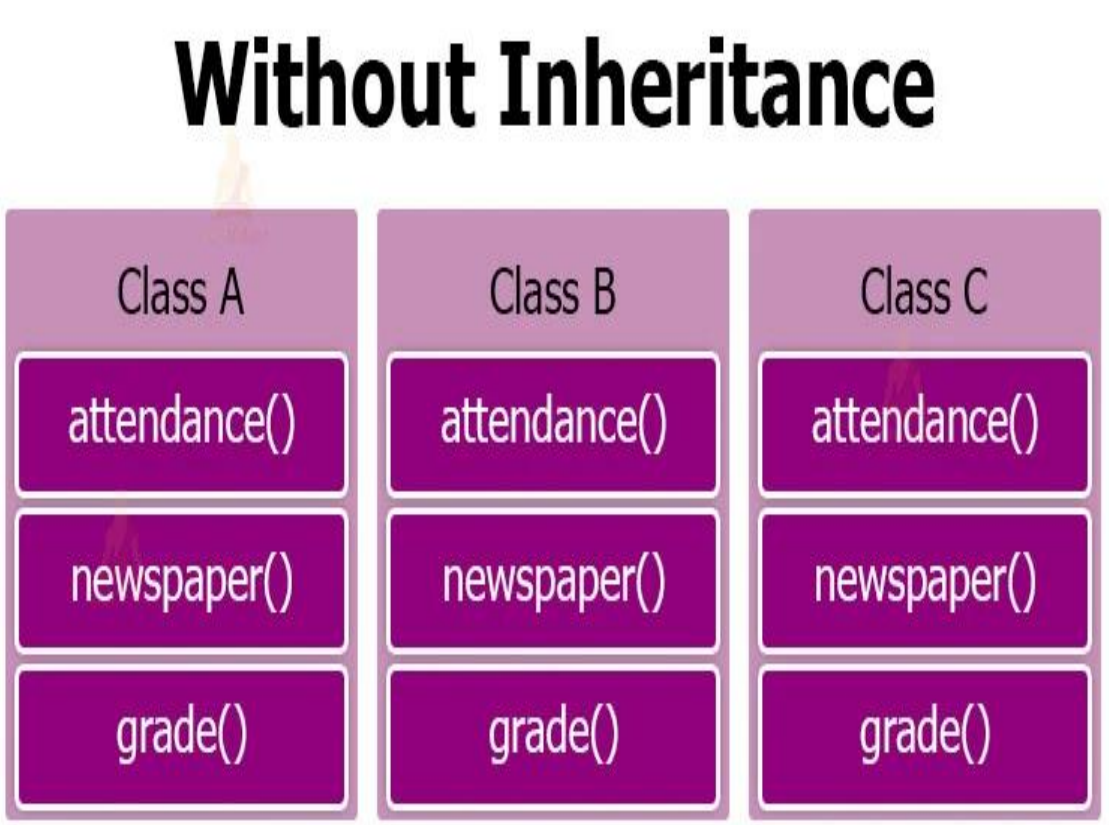
```
#include <iostream>
using namespace std;
class A {
private:
int a=10;
friend int fun(int x);
};
int fun(int x){
A obj;
return obj.a + x;
}
int main() {
cout<<“friend function access private value of a:”<<fun(44)<<endl;
return 0;
}
```

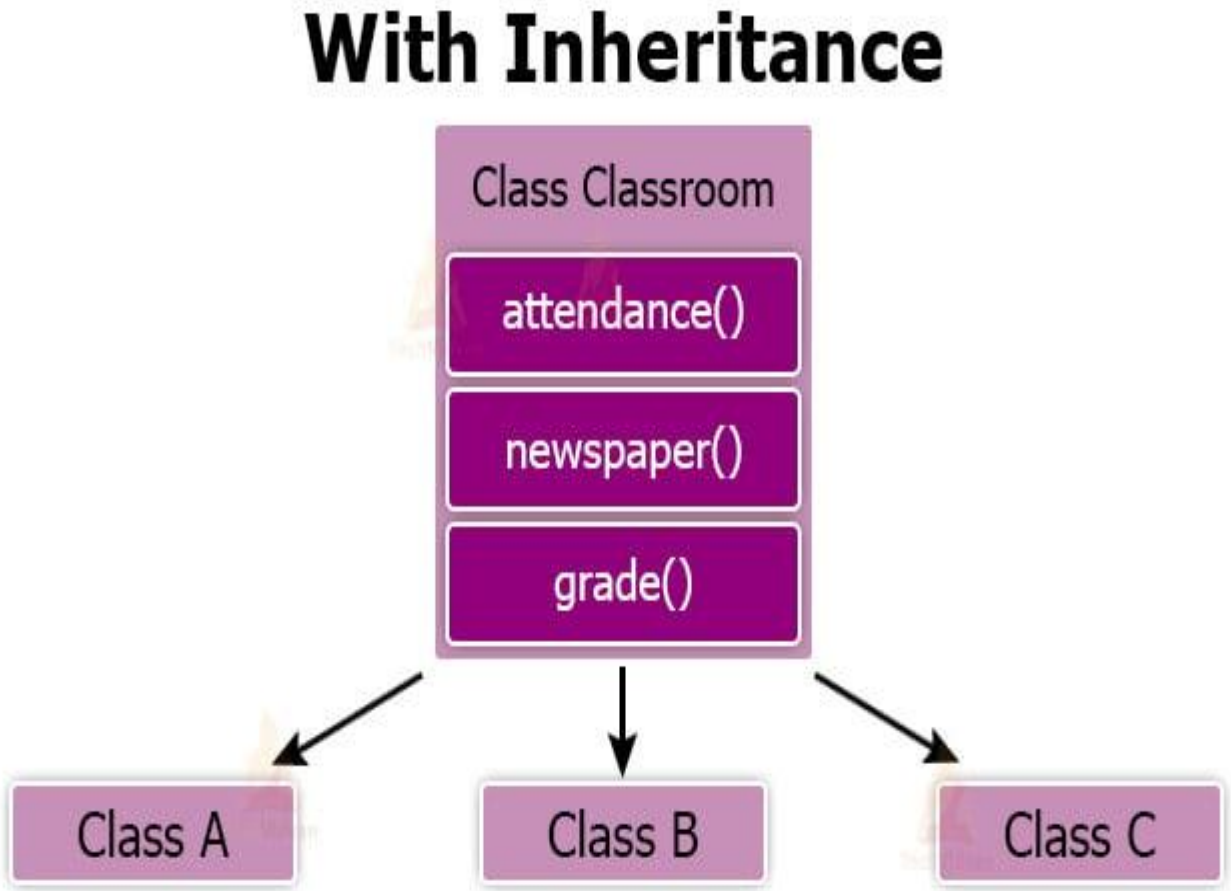
Inheritance

- Inheritance is a core concept of Object-Oriented Programming
- It allows a new (derived) class to inherit properties and behaviours from an existing class.
- It is capability of a class to derive property of another class.
- It provide code reusability, extensibility and hierarchical relationships among classes.
- It establishes an "is-a" relationship between classes.
- It allow us to defined a class in terms of another class.
- Existing class is called base class.
- New class is called derived class.

```
class Base
        {   // Base class members};
class Derived : public Base
                {   // Derived class members};
```

Inheritance





Inheritance

- **Base Class (Parent Class):**The class from which properties and methods are inherited.
- **Derived Class (Child Class):** The class that inherits the properties and methods from the base class.
- **Access Specifiers:** public, protected, and private. It used to control the visibility and accessibility of base class members within the derived class. We generally used public inheritance.

	Derived Class	Derived Class	Derived Class
Base Class	**Private Mode**	**Protected Mode**	**Public Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

Modes of Inheritance

1. **Public inheritance**
- The base class's public members become →public and
- protected members become →protected in the derived class.
- Private members are inaccessible.

2. Protected Inheritance
- base class's public & protected members become →protected in derived.
- Private members are inaccessible.

3. Private Inheritance
- Both public & protected members of base class become →private in derived.
- Private members are inaccessible.

Benefits of Inheritance

- **Code Reusability**: Reduces redundant code by allowing derived classes to reuse code from base classes.
- **Extensibility:** Allows for the creation of new classes that build upon existing functionality without modifying the original code.
- **Polymorphism:** Enables the use of base class pointers to refer to derived class objects, facilitating dynamic method dispatch.
- **Modularity:** Promotes a more organized and structured codebase.
- **Reliability:** By reusing a base class that has already been tested, the reliability of the new code is enhanced.
- Easy to understand.
- Save time & effort.

Types of Inheritance

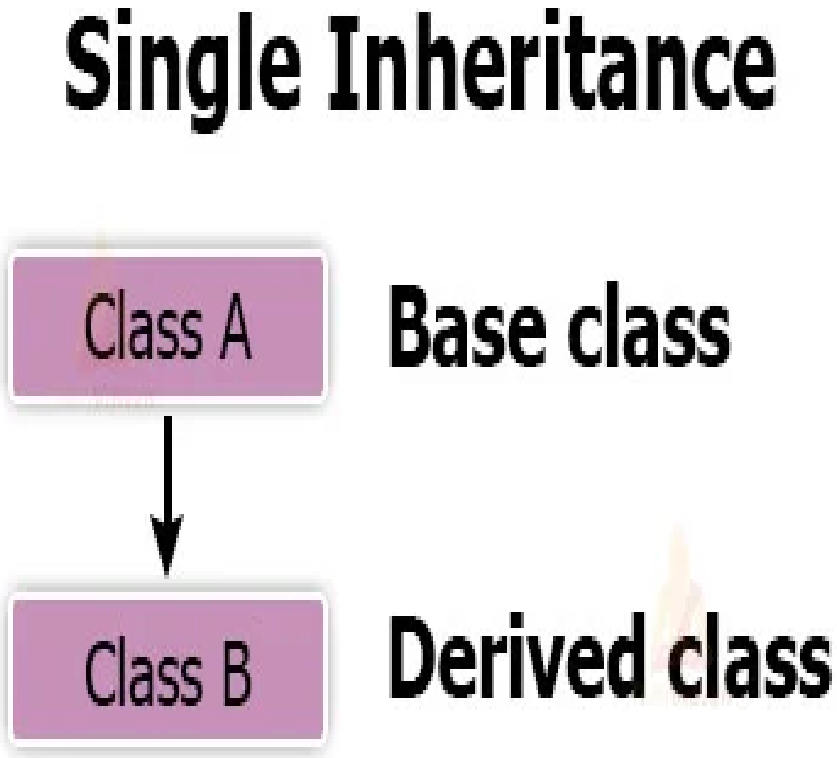
➢ **There are 5 types of Inheritance:**

1. **Single Inheritance:** A derived class inherits from only one base class.
2. **Multiple Inheritance:** A derived class inherits from multiple base classes.
3. **Multilevel Inheritance:** A derived class inherits from a base class, which itself is derived from another base class.
4. **Hierarchical Inheritance:** Multiple derived classes inherit from a single base class.
5. **Hybrid Inheritance:** A combination of two or more types of inheritance.

Single Inheritance

- In this type of inheritance, there is only one derived class inherited from one base class.

```
class base {
        //Body
      };
class derived : access_specifier base
    {
        //Body
    };
```



Single Inheritance

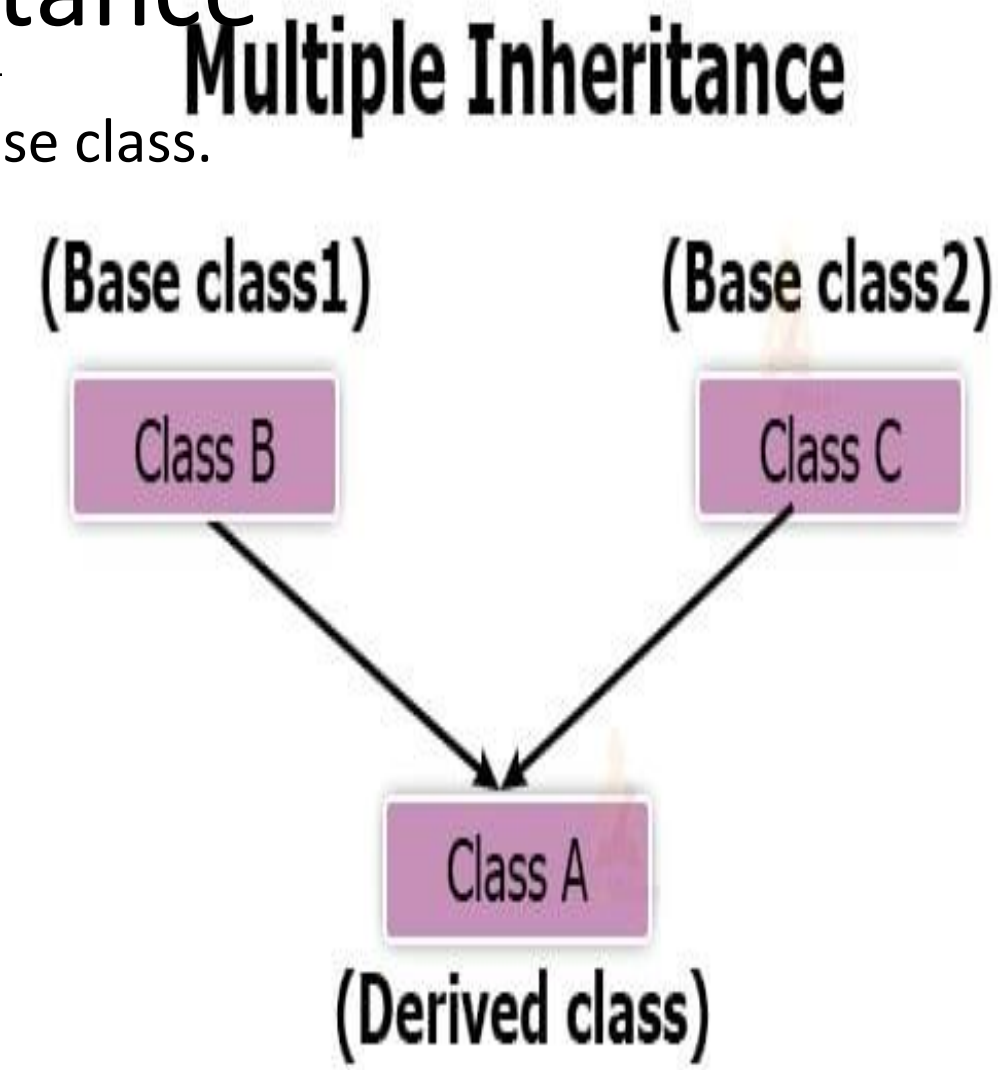
```
#include <iostream>
using namespace std;
class A
{
   public:
      void fun1()
       {
         cout<<“Base Class"<<endl;
       }
};
```

```
class B : public A
{
    public:
       void fun2() {
          cout<<“child class"<<endl;
       }   };
int main() {
  B obj;
  obj.fun1();
  obj.fun2();
  return 0;     }
```

Multiple Inheritance

- one derived class inherits from two or more base class.

```
class base1
{
      //body
 };
 class base2 {
                    //body
                 };
class derived : public base1, public base2
      {
         //body of derived class
       };
```



Multiple Inheritance

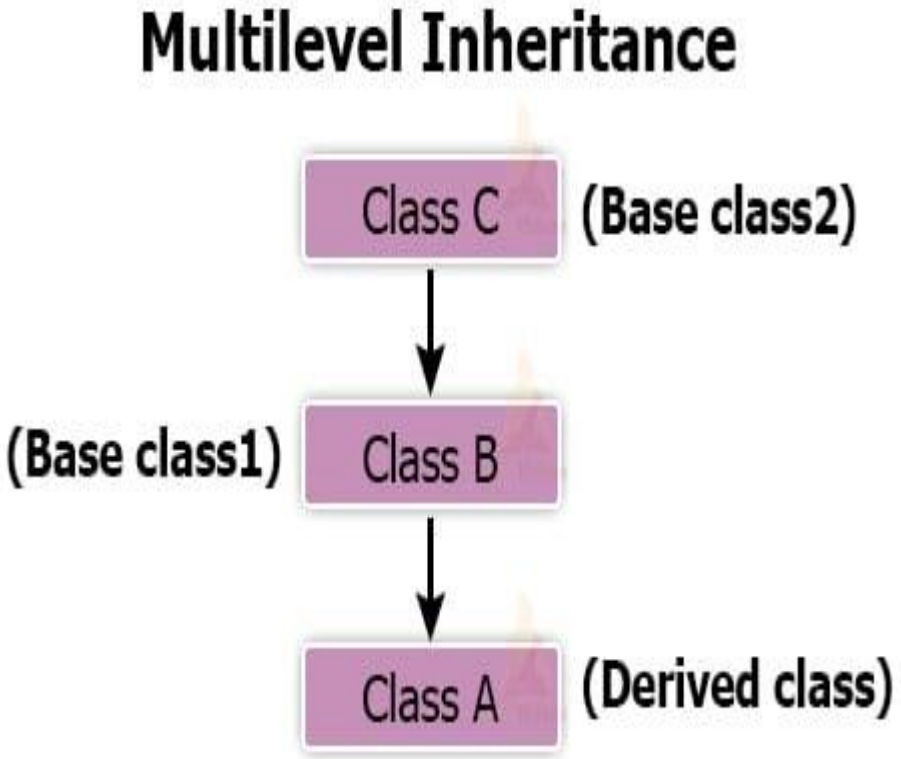
```
class A {
public:
void fun1( )
        {   cout<<“class A”;  }
          };
class B {
public:
void fun2( )
      {    cout<<“class B”;   }   };
```

```
class C : public A, public B
{
public:
void fun3( ) {
fun1( );
fun2( );    }    };
int main() {
C obj;
obj.fun1( );
obj.fun2( );
obj.fun3( );
return 0;   }
```

Multilevel Inheritance

- A derived class inherits from another derived class in this type of inheritance.

```
class C
        {  //body  };
class B : public C
                {    //body   };
class A : public B
                {  //Derived body   };
```



Multilevel Inheritance

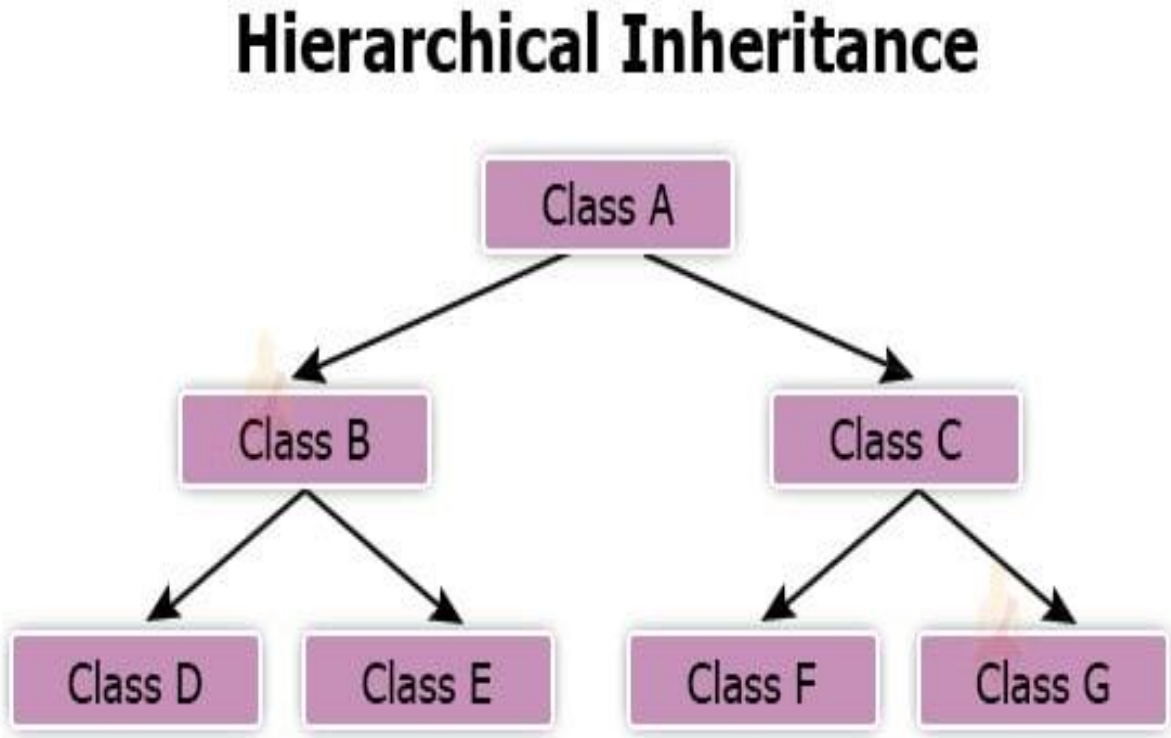
```
class A {
    public:
    void fun1() {
        cout<<“class A"<<endl;   }    };
class B: public A {
    public:
    void fun2() {
        cout<<“class B"<<endl;   }    };
```

```
class C : public B {
public:
void fun3() {
fun1();
fun2();
cout<<“class C"<<endl;
}    };
int main() {
C obj;
obj.fun3();
return 0;  }
```

Hierarchical Inheritance

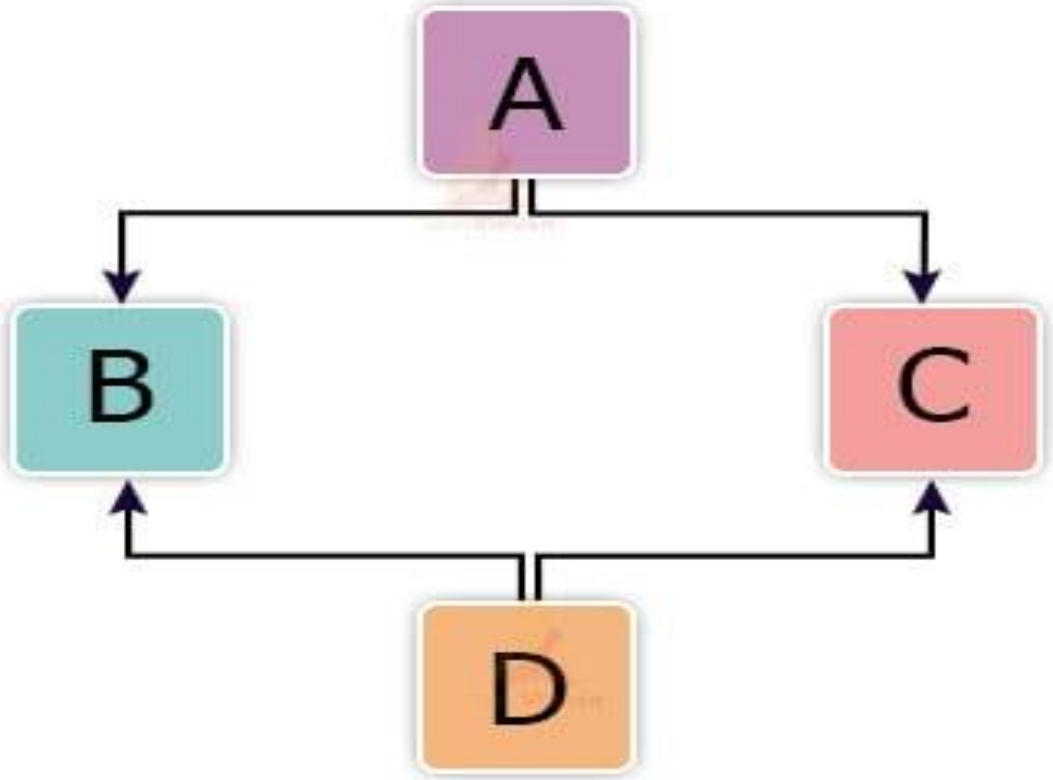
- Multiple derived classes are inherited from a single base class in this type of inheritance.

```
class base { //body };
class derived1 : public base
{  //body   };
class derived2 : public base
{  //body };
```



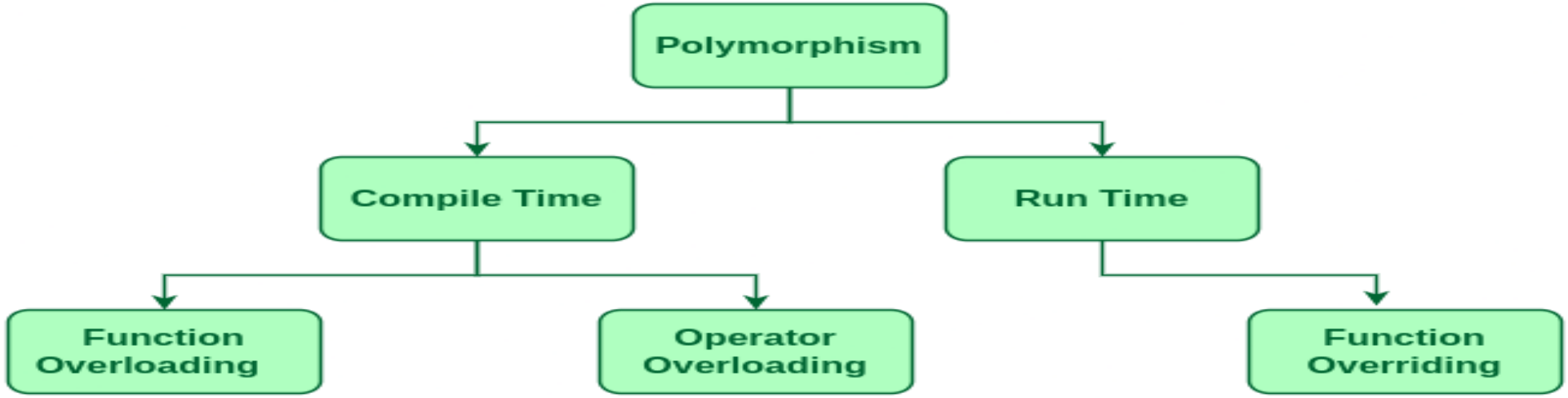
Hybrid Inheritance

- Hybrid inheritance is the combination of two or more types of inheritance. We can make various combinations in hybrid inheritance.



Polymorphism

- Polymorphism means same function possesses different behaviour in different situations.it is Important feature in oops
- Polymorphism, meaning having “many forms," .
- It enables code to be more flexible, reusable, and maintainable.
- It is ability of a message to be displayed in more than one form.
- Example boy at the same time son, sibling, friend, student.



Function Overloading

- **Compile-time Polymorphism/ Static Polymorphism / Early binding.**
- This type of polymorphism is **resolved during the compilation phase**.
- The compiler determines which function or operator to call based on the arguments' types and number.
- Allows multiple functions with the same name but different parameters.
- Function overloading is a feature of object-oriented programming
- where two or more functions can have the same name but behave differently for different parameters. Functions can be overloaded either by changing the number of arguments or changing the type of arguments.
- Types of Compile-Time Polymorphism

1. function overloading
2. operator overloading.

Rules for Function Overloading

1. The overloaded functions may differ by number of parameters.
2. The overloaded functions may differ by data types.
3. The same function name is used for various instances of function call.

- For example, if there is a function sum which performs addition operation then can use overloading functions like this –

```
int sum(int a,int b);
int sum(int a, int b,int c);
int sum(int a, int b,int c,int d);
```

- Similarly , the function overloading can be achieved like this –

```
int sum(int a, int b);
float sum(float a,float b);
double sum(double a, double b);
char sum(char a,char b);
```

- That means we can handle different number of parameters or different typ parameters using the same function name.

Function Overloading

```
class Addition {
public:
void add(int a, int b) {
        cout << "Integer Sum = " << a + b << endl;  }
void add( ) {
        cout << “Addition function ";    }    };
int main() {
    Addition A1;

    A1.add(10, 2);
    A1.add();
    return 0;    }
```

Function Overloading

```
class A {
        public:
             int add(int a, int b) {
                                        return a + b;       }
              string add(string a, string b) {
                                                return a + b;       }    };
int main() {
                 A obj;
                 cout << obj.add(5, 10);
                 cout << endl;
                 cout << obj.add("hello , ", "Chetna .!");
return 0;    }
```

Operator Overloading

- Operator Overloading Allows operators (like +, -, *, /) to be redefined in class.
- This provides a more intuitive way to perform operations on user-defined types.
- ability to provide the operators with a special meaning for particular data type, this ability is known as operator overloading.
- For example, we can make use of the addition **operator (+)** for string to concatenate two strings and for integer to add two integers.
- The **<< and >>** operator are binary shift operators but are also used with input and output streams. This is possible due to operator overloading.

Operator Overloading

- Operator overloading can be defined as an ability to define a new meaning for an existing (built-in) "operator".
- Various types of operators are overloaded:

1. Mathematical operators such as + ++
2. Relational operators such as < > ==
3. Logical operators such as && ||
4. Access operators [] ->
5. Assignment operator =
6. Stream I/O operators < < > >

- All of these operators have a predefined and unchangeable meaning for the built-in types.
- All of these operators can be given a specific interpretation for different classes or combination of classes.
- The overloading function can be a member or a non member function.

Operator Overloading

- C++ provides the flexibility to the programmers in extending these built-in operators.
- Define a function with keyword operator.
- Then write the operator as a function name. That means we can program that specified operator.
- Overloaded operator must be either Non static member function of class or At least one parameter should be class.
- Makes no assumptions about similar operators. For example, the fact that you overloaded + does not mean that you have also defined += for your class type.
- **Restrictions on use of operators.**
- It's not possible to change an operator's precedence.
- It's not possible to create new operators,
- For example: You can not redefine :: sizeof. ?, or. (dot).

Operator Overloading

```
class A {
        public:
                int x;
                A(int a) { x = a; }
                A operator+(A o)
                                { return A(x + o.x); }};
 int main() {
                A a1(5), a2(10);
                cout << (a1 + a2).x;
                return 0; }
```

Output:

15

Operator Overloading

```
#include <iostream>
using namespace std;
class A {
public:
    int x;
    A(int a)  { x=a;}
    // overload of +
    A operator+(A o) {
        return A(x + o.x);
    }  };
int main() {
    A a1(5), a2(10);
    A sum = a1 + a2;  // overload operator+
    cout<<sum.x;      // print “15”
    return 0;
}
```

Operator Overloading

```
class Complex {
public:
    int real, imag;
    Complex(int r, int i) { real = r; imag = i; }

Complex operator + (Complex c) {
        return Complex(real + c.real, imag + c.imag);
    }
};
int main() {
    Complex c1(2, 3), c2(4, 5);
    Complex c3 = c1 + c2;  // calls operator+
    cout << c3.real << " + " << c3.imag << "i" << endl;
    return 0;    }
```

Operator Overloading

```
class A{
public:
int a;
A(int x){        a = x;}
void operator !(){   a = -a;} };
int main(){
A ob(10);
cout<<"given value of a is : " <<ob.a;
!ob;    //! operator overload
cout<<"\nNew value operator overloading ! is : "<<ob.a;
return 0;}
```

Run-time Polymorphism

- Also called Dynamic Polymorphism function overriding.
- This type of polymorphism is resolved during program execution.
- It relies on inheritance and virtual functions.
- **Virtual Functions:** Declared in a base class using the virtual keyword and overridden in derived classes. When a base class pointer or reference points to a derived class object, calling a virtual function through that pointer/reference will invoke the derived class's version of the function. This is achieved through a mechanism called the "virtual table" (vtable).

LNCT GROUP OF COLLEGES
"WORKING TOWARDS BEING THE BEST"

Virtual Function

- Virtual function overwrite in derived class.
- It tells compiler to perform late binding on this function.
- Virtual keyword is used to defined virtual function.
- Virtual function ensure that the correct function is called for an object.
- Reference pointer used to call virtual function.
- A virtual function is a member function.
- It is declared within a base class and redefined by a derived class.
- The virtual function within the base class defines the form of the interface to that function Each redefinition of the virtual function by a derived class indicate some different task related to derived class.
- When a virtual function is redefined by a derived class, the keyword virtual is not needed. The virtual function written in base class acts as interface and the function defined in derived classes act as different forms of the same function. This property of virtual function brings the runtime polymorphism.

Run-time Polymorphism

```
class Base {
                public:
                virtual void show() { cout << "Base show\n"; }
            };
class Derived : public Base {
                                    public:
                                    void show() { cout << "Derived show\n"; }
                                };
int main() {
                    Base* b = new Derived();
                    b->show();
                    delete b;
                    return 0;
}
```

Output:
Derived show

Virtual Function

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

class Base {
public:
    virtual void show()
{
        cout << "Base class
show method." << endl;
    }
};
```

```
class Derived : public Base
{
public:
    void show()
{
        cout << "Derived class
method." << endl;
    }
};
```

```
int main()
{
    Base baseObj;
    Derived derivedObj;

    baseObj.show();

    derivedObj.show();

    return 0;

}
```

Virtual Function

```
class Shape {
public:
virtual void draw() {          cout << "Drawing a shape.";       }   };
class Circle : public Shape {
public:
void draw() {   cout << "Drawing a circle.";       }   };
int main() {
Shape* s1 = new Circle();
s1->draw(); // Calls Circle's draw()
delete s1;
return 0;    }
```

Compile-Time Polymorphism	Run-Time Polymorphism
It is also called Static Polymorphism.	It is also known as Dynamic Polymorphism.
In compile-time polymorphism, the compiler determines which function or operation to call based on the number, types, and order of arguments.	In run-time polymorphism, the decision of which function to call is determined at runtime based on the actual object type rather than the reference or pointer type.
Function calls are statically binded.	Function calls are dynamically binded.
Compile-time Polymorphism can be exhibited by: 1. Function Overloading 2. Operator Overloading	Run-time Polymorphism can be exhibited by Function Overriding.
Faster execution rate.	Comparatively slower execution rate.
Inheritance in not involved.	Involves inheritance.

Introduction to Data Structures

- A data structure represents the logical relationship that exists between individual elements of data to carry out certain task.
- A data structure defines a way of organizing all data items that consider not only the elements stored but also stores the relationship between the elements.
- Data structure deals with the study of how the data is organised in memory, how efficiently the data can be retrieved and manipulated and possible ways in which different data items are logically related.
- Data structure is collection of elements and all the possible operations which are required for those set of elements.
- A data structure in a set of domains D, a set of functions F and set of axioms A. This triple (D, F, A) denotes the data structure d.

Introduction to Data Structures

- A data structure is a specific way of organize, store, and manage data in a computer's memory to facilitate efficient access and modification.
- It provides a systematic approach to handling data, which is crucial for developing effective and scalable software.
- **Organization**: Data structures define how data elements are arranged relative to each other, establishing relationships and enabling logical access patterns.
- **Storage:** They dictate how data is physically stored in memory, influencing memory usage and retrieval speed.
- **Operations: I**t support various operations, such as insertion, deletion, searching, and traversal, which are performed on the stored data. The efficiency of these operations is a primary consideration when choosing a data structure.

Types of Data Structures

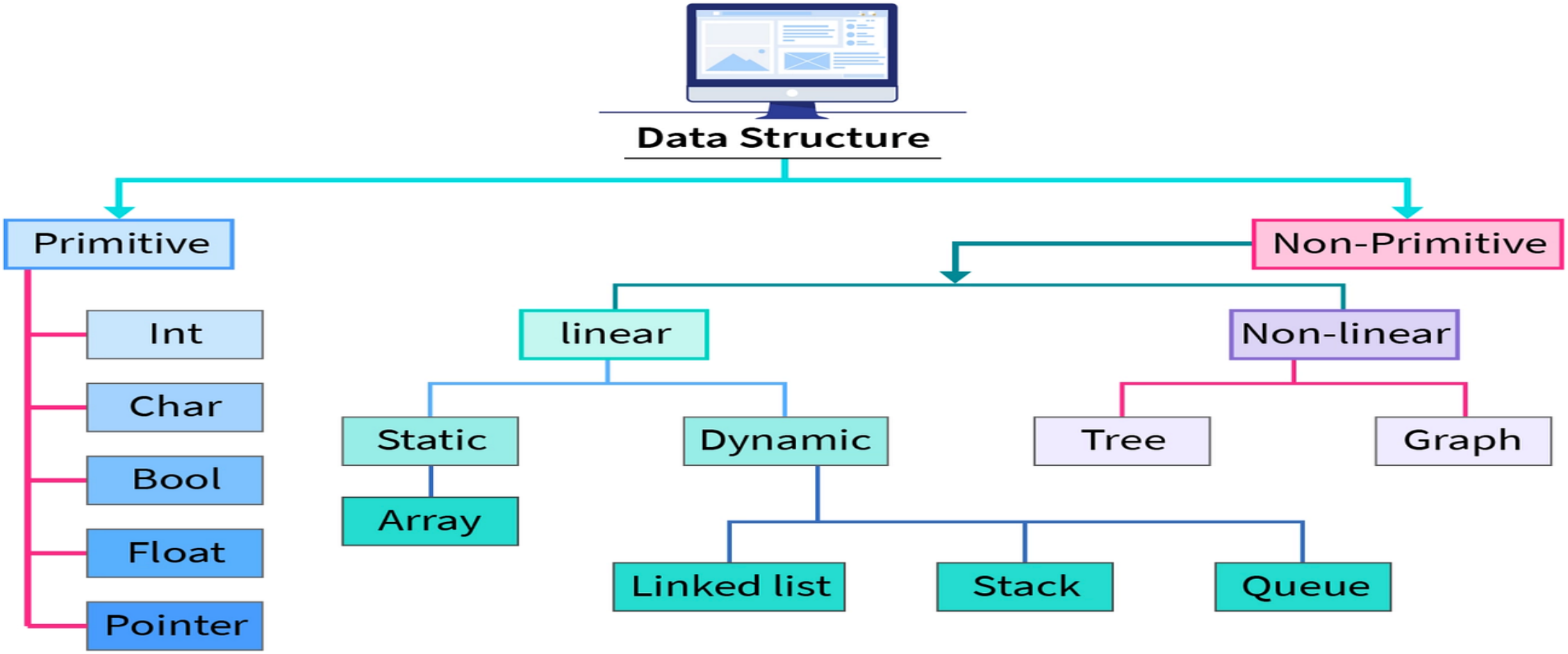
1. primitive data structure
2. Non-Primitive data structure.

- 1. Primitive data structures :- example: int, char, float
- 2. Non primitive data structures
- 2. (i) Linear data structures :- In linear data structures, data elements are arranged sequentially, one after the other. Each element has a unique predecessor (except for the first element) and a unique successor (except for the last element). where each element is attached to its previous and next adjacent elements.

a) Sequential Organization: Order Preservation
b) Fixed or Dynamic Size
c) Efficient Access
d) example: lists, stacks, queues

- 2. (ii) Non linear data structures :- Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. Examples are trees and graphs. example: trees, graphs

Types of Data Structures



Linear Data Structures

- Elements are arranged sequentially.
- Elements are arranged in one dimension.
- Example: Array, Linked lists, stack, queue, etc.

1. **Arrays:** A collection of elements of the same data type stored in contiguous memory locations.
2. **Linked Lists:** A sequence of nodes, where each node contains data and a pointer to the next node.
3. **Stacks:** A Last-In, First-Out (LIFO) structure where elements are added and removed from the same end.
4. **Queues:** A First-In, First-Out (FIFO) structure where elements are added at one end and removed from the other.

Applications of Data Structures

1. **Databases:** Data structures are used to organize and store data in a database, allowing for efficient retrieval and manipulation.
2. **Operating systems:** Data structures are used in the design and implementation of operating systems to manage system resources, such as memory and files.
3. **Computer graphics:** Data structures are used to represent geometric shapes and other graphical elements in computer graphics applications.
4. **Artificial intelligence:** Data structures are used to represent knowledge and information in artificial intelligence systems.

➢ **Advantages** of Data Structures:

1. **Efficiency:** Data structures allow for efficient storage and retrieval of data, which is important in applications where performance is critical.
2. **Flexibility:** Data structures provide a flexible way to organize and store data, allowing for easy modification and manipulation.
3. **Reusability:** Data structures can be used in multiple programs and applications, reducing the need for redundant code.
4. **Maintainability:** Well-designed data structures can make programs easier to understand, modify, and maintain over time.

Operations on Data Structures

1. **Traversing:** Basically to process a data structure if every element of data structure is visited once and only once, such type of operation is called as traversing.
2. **Insertion:** When an element of the same type is added to an existing data structure, the operation we are doing is called as Insertion operation. The element can be added anywhere in the data structure in the data structure.
3. **Deletion:** When an element is removed from the data structure, the operation we are doing is called as Deletion operation. We can delete an element from data structure from any position.
4. **Searching:** When an element is checked for its presence in a data structure, that operation we are doing is called as 'searching' operation. The element that is to be searched is called as key element.
5. **Sorting:** When all the elements of array are arranged in either ascending or descending order, the operation used to do this process is called as Sorting
6. **Merging:** When two lists List A and List B of size M and N respectively, of same type of elements, clubbed or joined to produce the third list, List C of size (M+N), and the operation done during the process is called as Merging.

Array [...]	Linked List
Elements are stored in contiguous memory locations	Elements are connected using pointers
Supports random access to elements	Only supports sequential access to elements
Insertions and deletions are tricky: elements need to be shifted	Insertions and deletions can be done efficiently without shifting
Fixed memory: static memory allocation	Dynamic memory allocation at runtime
Elements are independent of each other	Each node points to the next node or both the next and the previous node

Non-Linear Data Structures

- Elements are not arranged sequentially, allowing for more complex relationships.
- Elements are arranged in one-many, many-one and many-many dimensions.
- Example: tree, graph, table, etc.
- Trees: Hierarchical structures where data is organized in a parent-child relationship.
- Graphs: Collections of nodes (vertices) connected by edges, representing relationships between entities.
- Hash Tables: Structures that map keys to values for efficient data retrieval.