RASHTRASANT TUKADOJI MAHARAJ NAGPUR UNIVERSITY, NAGPUR

FOUR YEAR BACHELOR OF TECHNOLOGY (B.Tech.) DEGREE COURSE

SEMESTER: FOURTH (C.B.C.S)

BRANCH: INFORMATION TECHNOLOGY

Subject Name: Object Oriented Programming C++ **Subject code: BTIT403P**

Load	Credits	College Assessment Marks	University Evaluation	Total Marks
2Hrs (Practical)	1	25	25	50

Aim: 1) To study Object Oriented Programming C++

2) Program implementation on various features of OOP language.

Prerequisite(s): Students are expected to be proficient in a C++ programming language.

Course Objectives

1	**To understand the basic concept of OOP**
2	**To learn the properties /features of OOP**
3	**Implement the concept of classes & objects, Inheritance, Polymorphism, file handling Using C++**
4	**To understand the concept of Exception handling & STL.**

Course Outcome:

At the end of this course students are able to:

CO1	**Understand the basic concept & the properties /features of OOP**
CO2	**Understand and implement the concept of constructor & destructor in c++**
CO3	**Implement concept Inheritance using C++**
CO4	**Understand and implement the concept polymorphism using C++**
CO5	**Implement file handling in C++**
CO6	**Understand concept of Exception handling & STL**

K. D. K. COLLEGE OF ENGINEERING, NAGPUR
IMPARTING TECHNICAL EDUCATION SINCE 1984
NBA Accredited - Five programs:(Civil Engg., Mechanical Engg., Electrical Engg. Comp. Sc. & Engg., Electronics & Telecomm. Engg.) Grade A, NAAC Accredite

---

Department of Information Technology
Semester: IV

Name of Subject: Object Oriented Programming C++

Subject Code: BTIT403P

Course Outcome:

At the end of this course students are able to:

CO1	Understand the basic concept & the properties /features of OOP
CO2	Understand and implement the concept of constructor & destructor in c++
CO3	Implement concept Inheritance using C++
CO4	Understand and implement the concept polymorphism using C++
CO5	Implement file handling in C++
CO6	Understand concept of Exception handling & STL

LIST OF PRACTICALS

1	Write a C++ program to implement the concept of class and object.
2	Write a C++ program to implement the concept of function overloading.
3	Write a C++ program to implement the types of constructor.
4	Write a C++ program to implement the concept of operator overloading.
5	Write a C++ program to implement the types of inheritance.
6	Write a C++ program to implement the concept of friend function.
7	Write a C++ program to implement the string operation in C++.
8	Write a C++ program to implement the concept of pointers for object.
9	Write a C++ program to implement the concept of file handling
10	Write a C++ program to implement the template function.

Prof. S.S. Khokle
Subject Teacher

Dr.S.P. Khandait
Head of Department

PRACTICAL NO 1

AIM: Write a C++ program to implement the concept of class and object.

THEORY:
Class:
A class is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.
For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range,* etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define the properties and behaviour of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage,* etc, and member functions can be *applying brakes*, *increasing speed,* etc.

But we cannot use the class as it is. We first have to create an object of the class to use its features. An Object is an instance of a Class.
Defining Class in C++
A class is defined in C++ using the keyword class followed by the name of the class. The following is the syntax:

class ClassName {
access_specifier: //
***Body of the class* };**
Here, the access specifier defines the level of access to the class's data members.
Example: class ThisClass { public: int var; void print() { cout << "Hello";
}
};

Object : We can create an object of the given class in the same way we declare the variables of any other inbuilt data type.

ClassName ObjectName; Example:
MyClass obj;

Program 1 :

```
#include    <iostream>
#include<string>
using namespace std;
class car
{  public:
string brand;   string model;   int year;   };   int main()   {   car carobj1;
carobj1.brand="BMW";   carobj1.model="x5";   carobj1.year=1994;   car
carobj2;             carobj2.brand="balleno";             carobj2.model="xz";
```

```
carobj2.year=1999;
cout<<carobj1.brand<<"\n"<<carobj1.model<<"\n"<<carobj1.year<<"\n";
cout<<carobj2.brand<<"\n"<<carobj2.model<<”\n”<<carobj2.year<<"\n";
return 0;  }
```

Output :
BMW
x5 1994
balleno
xz
1999 Program
2 :

```
#include <iostream>
#include <string>
using namespace std;
class Person {  public:
string name;  int age;
void introduce ()
{
cout << "Hi, my name is " << name << " and I am "
<< age << " years old." << endl;
}  };  int
main()
{
Person person1;
person1.name = "XYZ";
person1.age = 30;
person1.introduce();  return
0;
}
```

OUTPUT:
Hi, my name is XYZ and I am 30 years old.
Conclusion: Hence, we performed this practical successfully

Viva Questions:
What is a class in C++?
What is an object in C++?
How do you define a class in C++?
How do you create an object of a class?
What is the difference between a class and an object?

PRACTICAL NO 2

AIM: Write a C++ program to implement the concept of function overloading.
THEORY: Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading "Function" name should be the same and the arguments should be different. Function overloading can be considered as an example of a polymorphism feature in C++.
If multiple functions have the same name but the parameters of the functions are different, it is known as Function Overloading.
If we have to perform only one operation and have the same name for the functions, it increases the program's readability. Suppose you have to perform the addition of the given numbers but there can be any number of arguments if you write the function such as a(int, int) for two parameters, and b(int,int, int) for three parameters then it may be difficult for you to understand the behavior of the function because its name differs.
Function overloading allows you to define multiple functions with the same name but different parameters.
The parameters should follow any one or more than one of the following conditions for Function overloading:
Parameters should have a different type

add(double a, double b) add
(int a,int b)

Program 1:

```
#include <iostream>
using namespace std; void
add(int a, int b)
{
  cout << "sum = " << (a + b);
}
void add(double a, double b)
{
    cout << endl << "sum = " << (a + b);
}
// Driver code int
main()
{     add(10,
2);
    add(5.3, 6.2);

    return 0;
}
```

Output *sum*
= 12 sum =
11.5

Viva Questions:

What is function overloading in C++?
Why do we need function overloading?
How does the compiler differentiate between overloaded functions?
What is the advantage of function overloading?
Can function overloading be achieved using default arguments?

PRACTICAL NO 3

AIM : Write a C++ program to implement the types of constructor.
THEORY : A constructor in C++ is a special member function with exact same name as the class name.
The constructor name is the same as the Class Name. Reason: Compiler uses this character to differentiate constructors from the other member functions of the class.A constructor must not declare a return type or void. Reason: As it's automatically called and generally used for initializing values.They can be defined inside or outside the class definition.Automatically calls when an object is created for the class.It uses to construct that means to initialize all the data members (variables) of the class.
Characteristics of Constructors in C++

- The name of the constructor is the same as its class name.
- Constructors are mostly declared in the public section of the class though they can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.

Definition Inside the class:
Syntax – class
CLASS_NAME
{
.... public
:
CLASS_NAME([parameter_list]) //constructor definition
{

....
}
//other member functions
};

Definition Outside the class:
Syntax –
class CLASS_NAME
{
.... public
:
CLASS_NAME([parameter_list]) //constructor Declaration
{
. . . . . .
}

//other member functions
};

CLASSNAME: :CLASSNAME([parameter_list]) //Constructor Definition
{
. . . .
}

Constructors types in C++

There are 3 types of Constructors in C++.
1. Default Constructor:
A constructor with no arguments (or parameters) in the definition is a default constructor. It is the type of constructor in C++ usually used to initialize data members (variables) with real values. Syntax of Default Constructor
className() { // body_of_constructor
}
The compiler automatically creates an implicit default constructor if the programmer does not define one.

2. Parameterized Constructor

Parameterized constructors make it possible to pass arguments to constructors.Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object. Syntax of Parameterized Constructor **className (parameters...) {**
// body
}

3. Copy Constructor

A copy constructor is a member function that initializes an object using another object of the same class.

Syntax of Copy Constructor

Copy constructor takes a reference to an object of the same class as an argument.
ClassName (ClassName &obj)
{
// body_containing_logic
}

1. Example of Defining the Constructor Within the Class

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

class Car { public:
string brand; string
model;
int year;

Car() { // Constructor defined within the class
brand = "Toyota";
model = "Camry";
year = 2022;

}
};
```

```
int main() { Car
myCar;
cout << "Brand: " << myCar.brand << endl; cout
<< "Model: " << myCar.model << endl; cout <<
"Year: " << myCar.year << endl;
return 0;
}
```

Output:
Brand: Toyota
Model: Camry
Year: 2022

2. Example Defining the Constructor Outside the Class

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

class Car {
public: string
brand; string
model;

int year;

Car(); // Constructor declaration
};

// Constructor definition outside the class
Car::Car() { brand =
"Ford"; model =
"Mustang"; year =
2020;
}

int main() { Car myCar; cout << "Brand: " <<
myCar.brand << endl; cout << "Model: " <<
myCar.model << endl; cout << "Year: " <<
myCar.year << endl;
return 0; }
```

Output:
Brand: Ford
Model: Mustang

Year: 2020

3. C++ Program to illustrate the concept of default constructors:

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

```
class Person {
public: string
name;
int age;

// Default constructor definition
Person() { name =
"John Doe"; age =
30;
}
};

int main() { Person personObj; cout << "Name:
" << personObj.name << endl;
cout << "Age: " << personObj.age << endl;

return 0; }
```

Output:
Name: John Doe
Age: 30

4. C++ Program to demonstrate the implicit default constructor

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

class Book {
public: string
title; string
author;
};

int main() { Book
bookObj;
cout << "Title: " << bookObj.title << endl; cout
<< "Author: " << bookObj.author << endl; return
0;
}
```

Output:
Title:
Author:

5. Defining Parameterized Constructor Inside The Class

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

```
class Rectangle {
private:            int
length;
int width;

public:
// Parameterized constructor defined inside the class
Rectangle(int l, int w) {
length = l; width = w;
}

int area() {
return length * width;

}
};

int main() {
Rectangle rect(5, 3); // Creating object with parameters
cout << "Area of Rectangle: " << rect.area() << endl;
return 0; }
```

Output:

Area of Rectangle: 15

Viva Questions:

What is a constructor in C++?
Why do we need constructors?
What are the different types of constructors in C++?
What is a default constructor?

PRACTICAL NO 4

AIM: Write a C++ program to implement the concept of operator overloading.

Theory: 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++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. Operator overloading is a compile-time polymorphism. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.

Example:

int a; float
b,sum; sum
= a + b;

Here, variables "a" and "b" are of types "int" and "float", which are built-in data types. Hence the addition operator '+' can easily add the contents of "a" and "b".
This is because the addition operator "+" is predefined to add variables of built-in data type only.

Can We Overload All Operators?

sizeof typeid
Scope resolution (::)
Class member access operators (.(dot), .* (pointer to member operator)) Ternary or conditional (?:)
Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded. Operators that can be Overloaded in C++
Unary operators
Binary operators

Special operators ([ ], (), etc)

Operators that can be overloaded Examples

Binary Arithmetic +, -, *, /, %
Unary Arithmetic +, -, ++, — Assignment
=, +=,*=, /=,-=, %=
Bitwise & , | , << , >> , ~ , ^
De-referencing (->)
Dynamic memory allocation,
De-allocation New, delete
Subscript [ ]
Function call ()
Logical &, | |, !
Relational >, < , = =, <=, >=

Important Points about Operator Overloading

1) For operator overloading to work, at least one of the operands must be a user-defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases (this behavior is the same as the copy constructor). See this for more details.

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

Types of Operator Overloading in C++

C++ provides a special function to change the current functionality of some operators within its class which is often called as operator overloading. Operator Overloading is the method by which we can change some specific operators' functions to do different tasks.

Syntax:

```
Return_Type classname :: operator op(Argument list)
{
Function Body
}
```

Operator Overloading can be done by using two approaches, i.e.
Overloading Unary Operator.
Overloading Binary Operator.

1. Overloading Unary Operator

Let us consider overloading (-) unary operator. In the unary operator function, no arguments should be passed. It works only with one class object. It is the overloading of an operator operating on a single operand.

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

class Distance { public:
int feet, inch;

Distance(int f, int i)
{ this->feet =
f; this->inch =
i;

}

void operator-()
{ feet--;
inch--;
cout << "\nFeet & Inches(Decrement): " <<
feet << "'" << inch;
}
};
```

```
int main()
{
Distance d1(8, 9);

-d1; return
0;
}
```

Output :
Feet & Inches(Decrement): 7'8

2. Overloading Binary Operator

In the binary operator overloading function, there should be one argument to be passed. It is the overloading of an operator operating on two operands. Below is the C++ program to show the overloading of the binary operator (+) using a class Distance with two distant objects.

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

class Distance { public:
int feet, inch;

Distance()
{
this->feet = 0; this->inch
= 0;
}

Distance(int f, int i)
{ this->feet =
f; this->inch =
i;
}

Distance operator+(Distance& d2)
{
Distance d3;

d3.feet = this->feet + d2.feet;
d3.inch = this->inch + d2.inch; return d3; }
};

int main()
{
Distance d1(8, 9);
Distance d2(10, 2);
Distance d3;
```

```
d3 = d1 + d2;

cout << "\nTotal Feet & Inches: " <<
d3.feet << "'" << d3.inch; return 0;
}
```

Output:
Total Feet & Inches: 18'11

Example of Operator Overloading in C++

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

class Complex { private:

int real, imag;

public:
Complex(int r = 0, int i = 0)
{ real = r;
imag = i;
}

Complex operator+(Complex const& obj)
{ Complex res; res.real = real
+ obj.real; res.imag = imag +
obj.imag;
return res; }
void print() { cout << real << " + i" << imag << '\n'; }
};

int main()
{
Complex c1(10, 5), c2(2, 4);
Complex c3 = c1 + c2;
c3.print();
}
```

Output:
12 + i9

Viva Questions:

What is operator overloading in C++?
Why do we need operator overloading?
Which keyword is used for operator overloading?
What are the rules for operator overloading?
Can all operators be overloaded? Which operators cannot be overloaded?
What is the syntax for overloading an operator

PRACTICAL NO 5

AIM: Write a C++ Program to implement the types of Inheritance

THEORY: A class's capability to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming in C++. In this article, we will learn about inheritance in C++, its modes and types, and how it affects different properties of the class. **Syntax of Inheritance in C++**

class derived_*class_name*: *access-specifier* base_*class_name* { *//body.* };

where,

- class: keyword to create a new class
- derived_class_name: name of the new class, which will inherit the base class
- access-specifier: Specifies the access mode which can be either of private, public or protected. If neither is specified, private is taken as default.
- base-class-name: name of the base class.

Modes of Inheritance in C++

The mode of inheritance controls the access level of the inherited members of the base class in the derived class. In C++, there are 3 modes of inheritance:

- Public Mode
- Protected Mode
- Private Mode

Public Inheritance 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.

Example:

class ABC : public XYZ {...}

Protected Inheritance Mode

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

Example:

class ABC: protected XYZ {...}

Private Inheritance Mode

If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become private in the derived class. They can only be accessed by the member functions of the derived class.

Private mode is the default mode that is applied when we don't specify any mode. **Example:**

class ABC : private XYZ {...} class ABC: XYZ {...}

Types Of Inheritance in C++

The inheritance can be classified on the basis of the relationship between the derived class and the base class. In C++, we have 5 types of inheritances:

1. **Single inheritance**
2. **Multilevel inheritance**
3. **Multiple inheritance**

4. **Hierarchical inheritance**
5. **Hybrid inheritance**

1.In single inheritance, a class is allowed to inherit from only one class. i.e. one base class is inherited by one derived class only.

Syntax
class *subclass_name* : *access_mode base_class* { *// body of subclass* }; Example:
Single Inheritance
class A { ... . }; class B: public A { ... . };

2. Multiple Inheritance
Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one subclass is inherited from more than one base class. Syntax
class *subclass_name* : access_mode *base_class1*, access_mode *base_class2*, ... { // *body of subclass* };

Here, the number of base classes will be separated by a comma (', ') and the access mode for every base class must be specified and can be different. **Multiple Inheritance**
class A { ... . }; class B { ... . }; class C: public A, public B { ... ... };

3. Multilevel Inheritance
In this type of inheritance, a derived class is created from another derived class and that derived class can be derived from a base class or any other derived class. There can be any number of levels.

Syntax class derived_class1: access_specifier base_class { ... . } class derived_class2: access_specifier derived_class1 { ... . } .....
Example:
Multilevel Inheritance
class C { ... . }; class B : public C { ... . }; class A: public B { ... ... };

4. Hierarchical Inheritance
In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class. Syntax class derived_class1: access_specifier base_class { ... . } class derived_class2: access_specifier base_class { ... . } Example:
Hierarchical Inheritance
class A { *// body of the class A.* } class B : public A { *// body of class B.* } class C : public A { *// body of class C.* } class D : public A { *// body of class D.* }

5. Hybrid Inheritance
Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance will create hybrid inheritance in C++
There is no particular syntax of hybrid inheritance. We can just combine two of the above inheritance types.
Example:
Below image shows one of the combinations of hierarchical and multiple inheritances:
Hybrid Inheritance

Class F
{
… … …
}
Class G
{
… … …
}
Class E: public F , Public G
{
…. …. …
}
Class A : Public B{
…. …. …
}
Class C : public B {
… …. …
}

Example 1: Program to Demonstrate the Simple Inheritance of a Class

```
#include   <iostream>
using  namespace  std;
class Parent {   public:
int id_p;
void printID_p()
{
cout << "Base ID: " << id_p << endl;
}    };    class  Child  :  public
Parent {  public:  int id_c;
void printID_c()
{
cout << "Child ID: " << id_c << endl;
}
};  int
main()  {
Child obj1;
obj1.id_p = 7;
obj1.printID_p();
obj1.id_c = 91;
obj1.printID_c()
return 0;  }
```

Output:
Base ID: 7
Child ID: 91

Example 2:
C++ program to demonstrate how to implement the Single Inheritance

```
#include
<iostream>        using
namespace  std;   class
Vehicle {  public:
```

```
Vehicle() { cout << "This is a Vehicle\n"; }
};  class Car : public Vehicle
{  public:
Car() { cout << "This Vehicle is Car\n"; }
};  int
main()  {
Car obj;  return
0;
}
```

Output:
This is a Vehicle
This Vehicle is Car

Example 3 :
C++ program to illustrate the multiple inheritance

```
#include   <iostream>
using  namespace  std;
class Vehicle {  public:
Vehicle() { cout << "This is a Vehicle\n"; }
};
class FourWheeler {  public:
FourWheeler() { cout << "This is a 4 Wheeler\n"; }
};
class Car : public Vehicle, public FourWheeler {
public:
Car() { cout << "This 4 Wheeler Vehical is a Car\n"; }
};  int
main()  {
Car obj;  return
0;
}
```

Output
This is a Vehicle
This is a 4 Wheeler
This 4 Wheeler Vehical is a Car

Example 4
C++ program to implement Multilevel Inheritance

```
#include   <iostream>
using  namespace  std;
class Vehicle {  public:
Vehicle() { cout << "This is a Vehicle\n"; }
};
class fourWheeler : public Vehicle {  public:
fourWheeler() { cout << "4 Wheeler Vehicles\n"; }
};
class Car : public fourWheeler {  public:
Car() { cout << "This 4 Wheeler Vehical is a Car\n"; }
```

```
};          int
main()     {
Car      obj;
return 0;
}
```

Output
This is a Vehicle
4 Wheeler Vehicles
This 4 Wheeler Vehical is a Car

Example 5
C++ program to implement Hierarchical Inheritance

```
#include   <iostream>
using  namespace  std;
class Vehicle {  public:
Vehicle() { cout << "This is a Vehicle\n"; }
};
class Car : public Vehicle {
public:
Car() { cout << "This Vehicle is Car\n"; }
};  class Bus : public Vehicle
{  public:
Bus() { cout << "This Vehicle is Bus\n"; }
};  int
main()  {
Car    obj1;
Bus    obj2;
return 0;
}
```

Output
This is a Vehicle
This Vehicle is Car
This is a Vehicle
This Vehicle is Bus

Example 6
C++ program to illustrate the implementation of Hybrid Inheritance

```
#include   <iostream>
using  namespace  std;
class Vehicle {  public:
Vehicle() { cout << "This is a Vehicle\n"; }
};  class Fare
{  public:
Fare() { cout << "Fare of Vehicle\n"; }
};
class Car : public Vehicle {  public:
Car() { cout << "This Vehical is a Car\n"; }
};
```

```
class Bus : public Vehicle, public Fare {  public:
Bus() { cout << "This Vehicle is a Bus with Fare\n"; }
};  int
main()  {
Bus obj2;  return
0;
}
```

Output
This is a Vehicle
Fare of Vehicle
This Vehicle is a Bus with Fare

Viva Questions:

What is inheritance in C++?
Why do we need inheritance?
What are the different types of inheritance in C++?
What is the syntax of defining inheritance in C++?
What is the difference between base class and derived class?

PRACTICAL NO 6

AIM: Write a c++ program to implement the concept of friend function.

THEORY: The friend function in C++ is defined outside the scope of the class. It has the authority to access all protected members and private members of the class. Friends are not member functions, but the prototypes for friend functions appear in the class function only. It covers class template, class, function template, function, and member functions, in which the entire members of the class are friends. **Syntax of Friend Function in C++**

You can declare the friend function using the keyword “friend” inside the body of the class.

class Box { double width; public: double length; friend void printWidth (Box box) ;
void setWidth (double wid) ;
} ;

Features of Friend Function in C++

- The friend function is invoked like a regular function without using the object and is declared in the public or private part.
- It is not in the scope of the class that declares it as a friend.
- It has to use the object name and dot membership operator with the member name to access the member names.

Advantages of Friend Function in C++

- The friend function allows the programmer to generate more efficient codes.
- It allows the sharing of private class information by a non-member function.
- It accesses the non-public members of a class easily.
- It is widely used in cases when two or more classes contain the interrelated members relative to other parts of the program.
- It allows additional functionality that is not used by the class commonly.

How to declare a Friend Function in C++?

All the friend class members become friend functions when a class is declared as a friend class. You can display the friend function in C++ in the following ways:

Class class_name
{
Friend data_type function_name(argument/s);
};

Another method of declaration can be:

class classB; class
classA {
// ClassB is a friend class of ClassA friend
class ClassB;
……
} class classB
{
……
}

The members of class A can be accessed from class B. The members of Class B can’t be accessed from inside Class A.

Program 1:

Adding members of two different classes using the friend function:

```
#include    <iostream>
using  namespace  std;
class   ClassB;     class
ClassA {  public:
ClassA()  :  numA(12)  {}
private:  int numA;
friend int add(ClassA, ClassB);
};  class ClassB
{  public:
ClassB()  :  numB(1)  {}
private:  int numB;
friend int add(ClassA, ClassB);
};
int add(ClassA objectA, ClassB objectB) {
return (objectA.numA + objectB.numB);
}
int      main()      {
ClassA   objectA;
ClassB objectB;
cout << "Sum: " << add(objectA, objectB);  return
0;
}
```

Output: Sum=13

Program 2:

Printing the length of a box using the friend function:

```
#include    <iostream>
using  namespace  std;
class  Box  {    double
width;  public:
friend void printWidth( Box box );
void setWidth( double wid );
};
void Box::setWidth( double wid ) {   width
= wid;
}
{
cout << "Width of box : " << box.width <<endl;
}   int main()
{    Box  box;
box.setWidt
h(10.0);
printWidth(
box );  return
0;
}
```

OUTPUT: Width of box : 10

Viva Questions:

What is a friend function in C++?
Why do we need a friend function?
How do you declare a friend function in a class?
Can a friend function access private and protected members of a class?
What is the syntax for defining a friend function?

PRACTICAL NO 7

AIM: Write a C++ program to implement the string operation in C++.

THEORY: In C++, strings are sequences of characters that are used to store words and text. They are also used to store data, such as numbers and other types of information in the form of text. Strings are provided by <string> header file in the form of std::string class.

Creating a String

Creating a string means creating an instance of std::string class as shown:

string str_name; where **str_name** is the name of the string.

Initializing a String

Initializing means assigning some initial value to the string. This can be done by using **assignment operator** and the text enclosed inside **"" double quotes.**

string str = "Some Text here";

The text inside "" is called **string literal** and it is the value that is assigned to the string variables.

Accessing String

A string can be referred using its name anywhere in the scope once it is declared. For example, the below example prints string using cout:

```
#include <iostream>  using
namespace std;
int main() {
string greeting = "Welcome to GfG!";
cout << greeting;  return 0;
}
```

Output

Welcome to GfG!

C++ String vs C Strings

The main difference between a string and a character array is that strings are immutable, while character arrays are not.

String	Character Array
Strings define objects that can be represented as string streams.	The null character terminates a character array of characters.
No Array decay occurs in strings as strings are represented as objects.	The threat of array decay is present in the case of the character array.
A string class provides numerous functions for manipulating strings.	Character arrays do not offer inbuilt functions to manipulate strings.
Memory is allocated dynamically.	The size of the character array has to be allocated statically.

C++ String Functions

C++ provides some inbuilt functions which are used for string manipulation, such as the strcpy() and strcat() functions for copying and concatenating strings. Some of them are:

Function	Description
length()	This function returns the length of the string.
swap()	This function is used to swap the values of 2 strings.
size()	Used to find the size of string
resize()	This function is used to resize the length of the string up to the given number of characters.
find()	Used to find the string which is passed in parameters
push_back()	This function is used to push the passed character at the end of the string
Function	Description
pop_back()	This function is used to pop the last character from the string
clear()	This function is used to remove all the elements of the string.
strncmp()	This function compares at most the first num bytes of both passed strings.
strncpy()	This function is similar to strcpy() function, except that at most n bytes of src are copied
strrchr()	This function locates the last occurrence of a character in the string.
strcat()	This function appends a copy of the source string to the end of the destination string
find()	This function is used to search for a certain substring inside a string and returns the position of the first character of the substring.
replace()	This function is used to replace each element in the range [first, last) that is equal to old value with new value.
substr()	This function is used to create a substring from a given string.
compare()	This function is used to compare two strings and returns the result in the form of an integer.
erase()	This function is used to remove a certain part of a string.
rfind()	This function is used to find the string’s last occurrence.

Program

```
#include     <iostream>
#include        <string>
using  namespace  std;
int main() {  string str1
= "Hello";  string str2 =
"World";                    //
Concatenation
string concatenated = str1 + " " + str2;
cout << "Concatenated String: " << concatenated << endl;
// Length of strings   cout << "Length of first string: " <<
str1.length() << endl;  cout << "Length of second string: " <<
str2.length() << endl;
// String Comparison  if
(str1 == str2) {
cout << "Both strings are equal." << endl;
} else {
cout << "Strings are not equal." << endl;
}
// Substring Extraction
cout << "First 3 characters of first string: " << str1.substr(0, 3) << endl;
return 0;  }}
```

OUTPUT:

Concatenated String: Hello World
Length of first string: 5
Length of second string: 5 Strings
are not equal.
First 3 characters of first string: Hel

Viva Questions:

What is a string in C++?
What are the different ways to declare a string in C++?
What is the difference between C-style strings (char array) and std::string?
How do you initialize a string in C++?
How can you take input for a string containing spaces?

PRACTICAL NO 8

AIM: Write a C++ program to implement concept of pointers for object.

THEORY: In C++, pointers allow direct memory access and manipulation, and they can be used with objects as well. Instead of working with object variables directly, we use object pointers to dynamically allocate memory, access members, and manage objects efficiently.

Using Pointers with Objects

1. Creating Object Pointers

An object pointer is a pointer that stores the address of an object.

ClassName *ptr;

Here, ptr is a pointer to an object of ClassName.

2. Accessing Members Using Object Pointers

When using an object pointer, we use the -> (arrow operator) instead of the dot . operator.

Example: ptr->memberFunction(); This is equivalent to:

(*ptr).memberFunction();

3. Dynamic Memory Allocation for Objects

We can create an object dynamically using the new keyword. ptr = new ClassName();

To release allocated memory, we use:

delete ptr;

PROGRAM

```
#include <iostream>
using namespace std;
class Student {
public:  string name;
int age;
Student(string n, int a) {
name = n;
age = a;  }
void display() {
cout << "Student Name: " << name << endl;
cout << "Student Age: " << age << endl;
}  };
int main() {
Student *ptr = new Student("John", 20);  cout
<< "Accessing through pointer:" << endl;  ptr-
>display();  delete ptr;  return 0;
}
```

Output:

Accessing through pointer:
Student Name: John
Student Age: 20

Viva Questions:

What is a pointer to an object in C++?
How do you declare a pointer to an object?
How do you access members of an object using a pointer?
What is the difference between . (dot operator) and -> (arrow operator)?
How do you dynamically allocate an object using a pointer?
How do you delete a dynamically allocated object?

PRACTICAL NO 9

AIM: Write a C++ program to implement the concept of file handling.

THEORY: Files are used to store data in a storage device permanently. File handling provides a mechanism to store the output of a program in a file and to perform various operations on it. A stream is an abstraction that represents a device on which operations of input and output are performed. A stream can be represented as a source or destination of characters of indefinite length depending on its usage.

In C++ we have a set of file handling methods. These include ifstream, ofstream, and fstream. These classes are derived from fstrembase and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in fstream and therefore we must include fstream and therefore we must include this file in any program that uses files.

In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream.

- ofstream: This Stream class signifies the output file stream and is applied to create files for writing information to files
- ifstream: This Stream class signifies the input file stream and is applied for reading information from files
- fstream: This Stream class can be used for both read and write from/to files.

All the above three classes are derived from fstreambase and from the corresponding iostream class and they are designed specifically to manage disk files.

C++ provides us with the following operations in File Handling:

- Creating a file: open()
- Reading data: read()
- Writing new data: write()
- Closing a file: close()

Moving on with article on File Handling in C++

Opening a File

Generally, the first operation performed on an object of one of these classes is to associate it to a real file. This procedure is known to open a file.

We can open a file using any one of the following methods:

1. First is bypassing the file name in constructor at the time of object creation.
2. Second is using the open() function.

To open a file use

1 open() function

Syntax

void open(const char* file_name,ios::openmode mode);

Here, the first argument of the open function defines the name and format of the file with the address of the file.

The second argument represents the mode in which the file has to be opened. The following modes are used as per the requirements.

Modes	*Description*
in	Opens the file to read(default for ifstream)
out	Opens the file to write(default for ofstream)
binary	Opens the file in binary mode
app	Opens the file and appends all the outputs at the end
ate	Opens the file and moves the control to the end of the file
trunc	Removes the data in the existing file
nocreate	Opens the file only if it already exists
noreplace	Opens the file only if it does not already exist

Example

```
1fstream new_file;
2      new_file.open("newfile.txt", ios::out);
```

In the above example, new_file is an object of type fstream, as we know fstream is a class so we need to create an object of this class to use its member functions. So we create new_file object and call open() function. Here we use out mode that allows us to open the file to write in it.

Default Open Modes :

- ifstream ios::in
- ofstream ios::out
- fstream ios::in | ios::out

PROGRAM 1:

opening/creating a file using the open() function

```
#include<iostream>
#include    <fstream>
using namespace std;
int main() {
fstream new_file;
new_file.open("new_file",ios::out);
if(!new_file)
{
cout<<"File creation failed";
}
else
{
cout<<"New file created";
new_file.close(); // Step 4: Closing file
}   return
0;
}
```

Output:

```
C:\Users\Lenovo\Desktop>g++ new_file.cpp -o new_file.exe

C:\Users\Lenovo\Desktop>new_file.exe
New file created
C:\Users\Lenovo\Desktop>
```

PROGRAM 2:

Writing to a File

```
#include   <iostream>
#include    <fstream>
using namespace std;
int main() {
fstream new_file;
new_file.open("new_file_write.txt",ios::out);   if(!new_file)
{
cout<<"File creation failed";
} else
{
cout<<"New file created";
new_file<<"Learning File handling";    //Writing to file new_file.close();
}    return
0;
}
```

Output:

```
C:\Users\Lenovo\Desktop>g++ new_file.cpp -o new_file.exe

C:\Users\Lenovo\Desktop>new_file.exe
New file created
C:\Users\Lenovo\Desktop>
```

PROGRAM 3:

Reading from a File

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

```
int main() {
fstream new_file;
new_file.open("new_file_write.txt",ios::in);    if(!new_file)
cout<<"No such file"; } else { char ch; while (!new_file.eof()) { new_file >>ch;  cout
<< ch;
}
new_file.close();     return
0;
}
```

Output:

```
LearningFilehandlingg[Finished in 1.7s]
```

PROGRAM 4:

Close a File

```
#include   <iostream>
#include    <fstream>
using namespace std;
int main() {
fstream new_file;
new_file.open("new_file.txt",ios::out);
new_file.close();     return
0;
}
```

Output:
The file gets closed.

Viva Questions:

What is file handling in programming?
What are the different types of files in programming?
What are the basic operations that can be performed on a file?
What is the purpose of fopen() and fclose()?
How do you read and write a file in C using fprintf() and fscanf()?

PRACTICAL NO 10

AIM: Write a C++ program to implement the template function.

THEORY: A function template starts with the keyword template followed by template parameter(s) inside <> which is followed by the function definition.

```
template <typename T>
T functionName(T parameter1, T parameter2, ...) {
   // code
}
```

In the above code, T is a template argument that accepts different data types (int, float, etc.), and typename is a keyword.
When an argument of a data type is passed to functionName(), the compiler generates a new version of functionName() for the given data type.

Calling a Function Template

Once we've declared and defined a function template, we can call it in other functions or templates (such as the main() function) with the following syntax

```
functionName<dataType>(parameter1, parameter2,...);
```

For example, let us consider a template that adds two numbers:

```
template <typename T> T
add(T num1, T num2) {
return (num1 + num2);
}
```

We can then call it in the main() function to add int and double numbers.

```
int main() {

   int          result1;
double result2;
   // calling with int parameters
result1 = add<int>(2, 3);    cout
<< result1 << endl;

   // calling with double parameters
result2 = add<double>(2.2, 3.3);
cout << result2 << endl;

   return 0;
}
```

PROGRAM:

Adding Two Numbers Using Function Templates

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

template <typename T> T
add(T num1, T num2) {
return (num1 + num2);
}

int main() {      int
result1;      double
result2;
   //    calling    with    int    parameters
result1 = add<int>(2, 3);
   cout << "2 + 3 = " << result1 << endl;

   // calling with double parameters
result2 = add<double>(2.2, 3.3);     cout <<
"2.2 + 3.3 = " << result2 << endl;

   return 0;
}
```

Output

2 + 3 = 5
2.2 + 3.3 = 5.5

Viva Questions:

What is a template in C++?
What is the difference between function templates and class templates?
Why do we use templates in C++?
How do you define a function template?

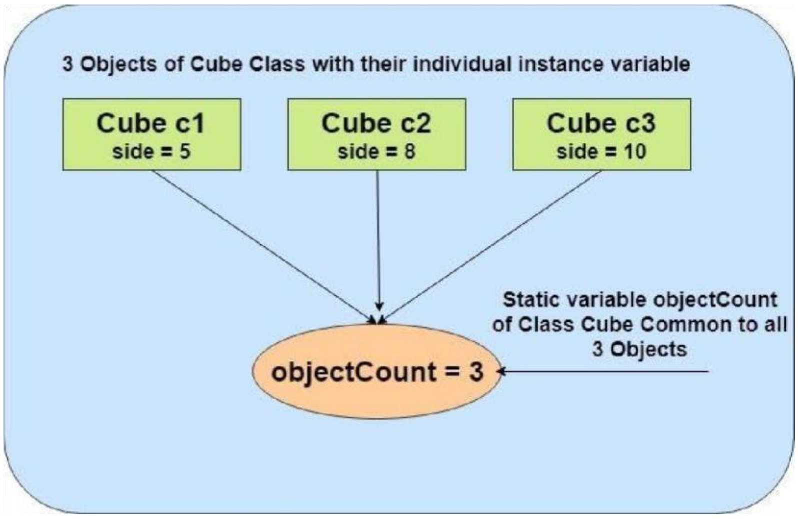
PRACTICAL NO 11

AIM: Write a c++ program to implement the concept of Static data and member function.

THEORY:A data member of a class can be qualified as static. The properties of a static member variable are similar to that of Cs static variable. A static data member has certain special characteristics. They are:-

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

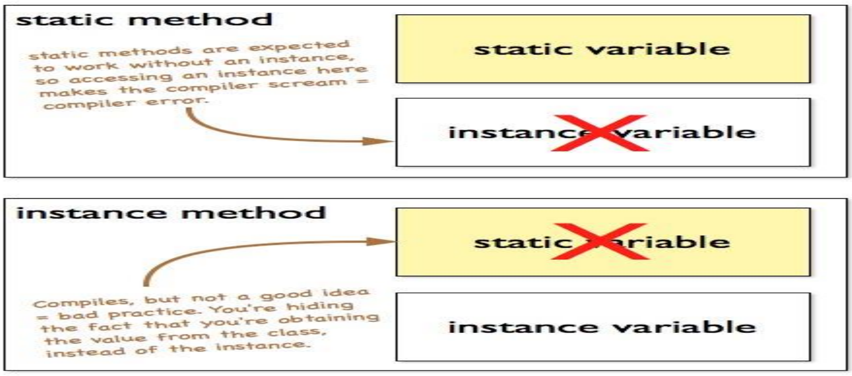
A static variable is normally used to maintain value common to the entire class. For e.g, to hold the count of objects created. Note that the type and scope of each static member variable must be declared outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object.



Static Member Function

like a static member variable, we can also have static member functions. A member function that is declared static has the following properties:-

- A static function can have access to only other static members (function or variable) declared in the same class.
- A static member function can be called using the class name (instead of its object) as follows- Class_name::Function_name();



PROGRAM:

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

class stat {
int code;
   static int count;

public:
    stat() {        code
= ++count;
   }

   void showcode() {
      cout << "\n\tObject number is :" << code;
   }

   static void showcount() {
      cout << "\n\tCount Objects :" << count;
   }
};

int stat::count;

void main() {
   clrscr();    stat
obj1, obj2;

   obj1.showcount();
obj1.showcode();
obj2.showcount();
obj2.showcode();
getch();
}
```

Output

Count Objects: 2
Object Number is: 1
Count Objects: 2
Object Number is: 2

Viva Questions:

What is a static data member in C++?
How is a static member function different from a regular member function?
What is the purpose of the static keyword in a class?
How do you define a static data member in a class?
How is a static data member initialized?

PRACTICAL NO 12

AIM: Write a C++ program to implement the concept of Inline Function without Class Using C++ Programming.

THEORY: An inline function in C++ is a function for which the compiler replaces the **function** call with the actual function code at compile time. This helps in reducing the overhead of function calls and improves execution speed.

In various versions of the C and C++ programming languages, an inline function is a function upon which the compiler has been requested to perform inline expansion. In other words, the programmer has requested that the compiler insert the complete body of the function in every place that the function is called, rather than generating code to call the function in the one place it is defined. Compilers are not obligated to respect this request.

Syntax Of Inline Function

```
inline return_type function_name(arguments...)
{
  //function_code
return_value;
}
```

PROGRAM:
Program for Inline Function without Class

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

using namespace std;

// Inline function without class
inline  float  cube(float  x)  {
return (x * x * x);
}

int main() {     float
val1, val2;

   cout  <<  "Enter  two  values:";
cin >> val1>>val2;

   cout << "\n\nCube value for val1 is          :" << cube(val1) << endl;
cout << "\n\nCube value for val2 is          :" << cube(val1) << endl;
getch();
}
```

Output
Enter two values:5
6

Cube value for val1 is :125
Cube value for val2 is :125

Viva Questions:

What is an inline function in C++?
How do you declare an inline function in C++?
Can you write an inline function outside a class?
What is the difference between a normal function and an inline function?
How does an inline function work at compile time?

PRACTICAL NO 13

AIM: Write a program to overload new/delete operators in a class.

THEORY: In C++, new and delete operators are used to dynamically allocate and deallocate memory. By default, these operators use the heap memory (free store) and call malloc() and free() internally.
However, C++ allows overloading these operators to customize memory allocation and deallocation for a class.

- To implement **custom memory management** (e.g., memory pools, logging, debugging).
- To optimize **memory allocation** for performance-critical applications.
- To track memory allocation (e.g., prevent memory leaks).
- To allocate memory from specific regions (e.g., shared memory, custom heap). **Syntax**

void* operator new(size_t size); // Overloads new void
operator delete(void* ptr); // Overloads delete

New Operator Overload

- operator new(size_t size) is a static function that takes the number of bytes to allocate.
   It returns a void* pointer to the allocated memory.

Delete Operator Overload

- operator delete(void* ptr) is a static function that takes a pointer to memory. It deallocates memory allocated using operator new.

PROGRAM:

```
#include <iostream>
#include <cstdlib>  // Required for malloc() and free() using
namespace std;

class CustomMemory { public:
   int data;

   //   Overloading   the   new   operator
void* operator new(size_t size) {
      cout << "Custom new called. Allocating " << size << " bytes.\n";
void* ptr = malloc(size);  // Allocating memory        if (!ptr) {
         cout   <<   "Memory   allocation   failed!"   <<   endl;
exit(1); // Exit if allocation fails
      }
      return ptr;
   }

   // Overloading the delete operator
   void operator delete(void* ptr) {
      cout << "Custom delete called. Deallocating memory.\n";
      free(ptr);  // Freeing memory
```

```
    }
};

int main() {
    // Using overloaded new operator
    CustomMemory* obj = new CustomMemory();       obj-
>data = 100;
    cout << "Data: " << obj->data << endl;

    // Using overloaded delete operator
    delete obj;

    return 0;
}
```

Output:

Custom new called. Allocating 4 bytes.
Data: 100
Custom delete called. Deallocating memory.

Viva Questions:

What is operator overloading in C++?
Why do we need to overload the new and delete operators?
What is the default behavior of new and delete in C++?
What is the syntax for overloading the new operator?
What is the syntax for overloading the delete operator?
How does memory allocation work in operator new()?

PRACTICAL NO 14

AIM: Write a C++ program to declare struct. Initialize and display contents of member variables.

THEORY: A struct (short for structure) in C++ is a user-defined data type that groups related variables of different data types into a single unit. It is similar to a class, but by default, all members in a struct are public.
A structure is declared using the struct keyword, followed by the structure name and member variables.

Syntax:

```
struct StructureName {
data_type      member1;
data_type      member2;
// More members...
};
```

Algorithm:

1. Define the structure consisting of members name , roll number and marks of the student 2. Read the student name, roll number and marks of the student.
3. Print the student name, roll number and marks of the student.

PROGRAM:

```
#include     <iostream>
using   namespace   std;
struct student
{
  char     name[50];
int  roll;          float
marks;
};

int main()
 {      student
s;
  cout << "Enter information," << endl;
cout << "Enter name: ";      cin >>
s.name;
  cout << "Enter roll number: ";        cin >>
s.roll;     cout << "Enter marks: ";
 cin >> s.marks;     cout << "\nDisplaying Information," << endl;
cout << "Name: " << s.name << endl;      cout << "Roll: " <<
s.roll << endl;      cout << "Marks: " << s.marks << endl;
return 0;
 }
```

Output :

Enter information, Enter name: Bill Enter roll number: 4 Enter marks: 55.6 Displaying Information, Name: Bill Roll: 4 Marks: 55.6

Viva Questions:

What is a struct in C++?
How is a struct different from a class?
What is the default access specifier in a struct?
How do you declare a structure in C++?
Can a struct have different data types as members?