Roll Number:

Practice-1

i)Aim: To write a C++ program that calculates the factorial of a given number using a recursive function.

Description: In this program, recursion is used to calculate the factorial of a given number. The factorial of a non-negative integer n is the product of all positive integers less than or equal to n (i.e., n! = n × (n-1) × (n-2) × … × 1) . A recursive function works by calling itself until a base condition is met. In this case, the base condition is when n becomes 0 or 1, where the factorial is defined as 1. For values greater than 1, the function multiplies n with the factorial of n-1. This process continues until the base case is reached.

Program:

```
#include <iostream>

using namespace std;

// Recursive function to calculate factorial

int factorial (int n) {

   // Base case: factorial of 0 or 1 is 1

   if (n == 0 || n == 1)

      return 1;

   else

      return n * factorial (n - 1) ;  // Recursive call

}

int main () {

   int number;

   cout  <<   "Enter a positive integer: ";

   cin >>  number;

   // Error handling for negative input

   if (number < 0) {

       cout << "Factorial is not defined for negative numbers." << endl;

   } else {

      int result = factorial(number);
```

```
        cout << "Factorial of " << number << " is: " << result << endl;
    }
    return 0;
}
```

Output:

Enter a positive integer: 5

Factorial of 5 is: 120

Roll Number:

ii)Aim : To write a c++ program that demonstrates concept of call by value and call by reference

Description: In Call by Value, a copy of the actual variable is passed to the function.

Any changes made inside the function are performed on the copy, not on the original variable.

Hence, the original value remains unchanged after the function call.

Program:

```
#include <iostream>

using namespace std;

// Function to modify value

void modify (int  x) {

   x = x + 5 ;   // change applied to the copy

}

int main() {

   int a = 10;

   cout << "Value of a before function call: " << a << endl ;

   modify(a) ;   // Call by Value

   cout << "Value of a after function call: " << a << endl;

   return 0;

}
```

Output

Value of a before function call :10

Value of a after function call : 10

Call by Reference:

```
#include <iostream>
using namespace std;
void swapNumbers(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
    cout << "Inside function (after swap): a = " << a << ", b = " << b << endl;
}

int main() {
    int x = 10, y = 20;
    cout << "Before function call: x = " << x << ", y = " << y << endl;
    // Call by reference
    swapNumbers(x, y);
    cout << "After function call: x = " << x << ", y = " << y << endl;
    return 0;
}
```

Roll Number:

Practice-2

1)Aim: To demonstrate the difference between global and local variables using the scope resolution operator (::).

Description: In C++, local variables take precedence over global variables inside functions. The scope resolution operator (::) is used to access the global variable when a local variable of the same name exists.

Program:

```
#include <iostream>
using namespace std;
int x = 100;  // Global variable
int main() {
    int x = 50;   // Local variable
    cout << "Local variable x = " << x << endl;
    cout << "Global variable x = " << ::x << endl; // using scope resolution
    return 0;
}
```

Output:

Local variable x = 50

Global variable x = 100

2)Aim: To demonstrate the use of custom namespaces in C++.

Description:

Namespaces allow organizing code and avoiding naming conflicts.

We can define custom namespaces and access their members using the scope resolution operator (::).

Program:

```
#include <iostream>
using namespace std;
// Custom namespace
namespace Math {
    int add(int a, int b) {
        return a + b;
    }
}
namespace Physics {
    double add(double a, double b) {
        return a + b;
    }
}
int main() {
    cout << "Sum using Math namespace: " << Math::add(5, 10) << endl;
    cout << "Sum using Physics namespace: " << Physics::add(12.5, 7.5) << endl;
    return 0;
}
```

Output:

Sum using Math namespace: 15 Sum using Physics namespace: 20

Roll Number:

Practice-3

Definition:

An inline function is a special type of function in C++ where the compiler replaces the function call with the actual body of the function during compilation. This avoids the overhead of function calls such as stack maintenance, jump, and return.

Syntax:

inline return_type function_name(parameters) {

}

Benefits of Inline Functions:

Eliminates function call overhead, making execution faster.

Useful for small, frequently used functions.

Improves performance in programs with repeated function calls.

Keeps code clean and modular while still being efficient.

When NOT to Use Inline Functions:

For large or complex functions (may increase executable size).

For recursive functions (compiler usually ignores inline).

When using static variables inside functions (unexpected behavior).

For functions stored in separate libraries (inlining may not work across multiple translation units).

Roll Number:

Program to Illustrate Inline Functions

1)Aim: To demonstrate the concept and working of inline functions in C++.

Description:

This program shows how inline functions can be used to calculate the square and cube of a number. Instead of calling the function in the traditional way, the compiler replaces the call with the actual code of the function. This reduces execution time. Inline functions are generally used for small mathematical operations or repetitive tasks where efficiency matters.

Program:

```
#include <iostream>
using namespace std;
// Inline functions
inline int square(int n) {
    return n * n;
}
inline int cube(int n) {
    return n * n * n;
}
int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    cout << "Square of " << num << " = " << square(num) << endl;
    cout << "Cube of " << num << " = " << cube(num) << endl;
    return 0;
}
```

Roll Number:

Input:

Enter a number: 6

Output:

Square of 6 = 36

Cube of 6 = 216

Roll Number:

Practice-4

Aim: 1)To model a bank account using classes, objects, and encapsulation.

Description:

Encapsulation is the process of wrapping data and methods together in a class.

In this program, a Bank Account class is created with private data members such as accountNumber, holderName, and balance.

Public member functions are provided for deposit, withdraw, and display to access and modify the private data.

Program:

```
#include <iostream>
using namespace std;
class BankAccount {
private:
    int accountNumber;
    string holderName;
    double balance;
public:
    // Constructor
    BankAccount(int accNo, string name, double bal) {
        accountNumber = accNo;
        holderName = name;
        balance = bal;
    }
        void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount << endl;
    }
```

```
    void withdraw(double amount) {
        if (amount > balance) {
            cout << "Insufficient Balance!" << endl;
        } else {
            balance -= amount;
            cout << "Withdrawn: " << amount << endl;
        }
    }
        void display() {
        cout << "Account Number: " << accountNumber << endl;
        cout << "Holder Name: " << holderName << endl;
        cout << "Balance: " << balance << endl;
    }
};
int main() {
    BankAccount acc1(101, "John Doe", 5000);
    acc1.display();
    acc1.deposit(2000);
    acc1.withdraw(3000);
    acc1.display();
    return 0;
}
```

Output:

Account Number: 101

Holder Name: John Doe

Balance: 5000

Deposited: 2000

Withdrawn: 3000

Account Number: 101

Holder Name: John Doe

Balance: 4000

Access Specifiers: Public vs Private

Aim:

ii)To demonstrate the use of public and private access specifiers in C++.

Description:

Private members can only be accessed inside the class.

Public members can be accessed from outside the class using objects.

This program shows how private variables are hidden and accessed only via public functions.

Program:

```
#include <iostream>
using namespace std;
class Student {
private:
   int rollNo;      // private
   string name;     // private
public:
   void setData(int r, string n) {
      rollNo = r;
      name = n;
   }
      void display() {
      cout << "Roll Number: " << rollNo << endl;
      cout << "Name: " << name << endl;
   }
};
int main() {
```

```
    Student s1;
    s1.rollNo = 10;
    s1.setData(10, "Alice");
    s1.display();
    return 0;
}
```

Output:

Roll Number: 10

Name: Alice

Roll Number:

Aim:

iii)To demonstrate the use of the this pointer in C++.

Description:

The this pointer refers to the current object of a class.

It is used when local variables and data members have the same name, to resolve naming conflicts.

Program:

```
#include <iostream>
using namespace std;
class Employee {
private:
    int id;
    string name;
public:
    void setData(int id, string name) {
        // Here local variables shadow the class members
        this->id = id;
        this->name = name;
    }
        void display() {
        cout << "Employee ID: " << id << endl;
        cout << "Employee Name: " << name << endl;
    }
};
int main() {
    Employee e1;
    e1.setData(101, "Robert");
```

```
    e1.display();
    return 0;
}
```

Output:

Employee ID: 101

Employee Name: Robert

Roll Number:

Practice-5

Aim:

i)To demonstrate function overloading in C++.

Description:

Function overloading allows multiple functions to have the same name but different parameter lists.

The compiler determines which function to call based on the number or type of arguments.

Program:

```
#include <iostream>
using namespace std;
class Math {
public:
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
     int add(int a, int b, int c) {
        return a + b + c;
    }
};

int main() {
    Math m;
    cout << "Sum of 2 integers: " << m.add(5, 10) << endl;
    cout << "Sum of 2 doubles: " << m.add(4.5, 3.2) << endl;
```

```
    cout << "Sum of 3 integers: " << m.add(1, 2, 3) << endl;


    return 0;
}
```

Output:

Sum of 2 integers: 15

Sum of 2 doubles: 7.7

Sum of 3 integers: 6

2. Default Arguments

Aim:

ii)To demonstrate the use of default arguments in C++.

Description:

Default arguments allow a function to be called without providing all parameters.

If arguments are not supplied, the default values are used.

Program:

```
#include <iostream>
using namespace std;
int power(int base, int exp = 2) {  // default exponent is 2
    int result = 1;
    for (int i = 0; i < exp; i++) {
        result *= base;
    }
    return result;
}
    int main() {
    cout << "5 raised to 2 (default) = " << power(5) << endl;
    cout << "5 raised to 3 = " << power(5, 3) << endl;
    return 0;
}
```

Output:

5 raised to 2 (default) = 25

5 raised to 3 = 125

3. Friend Function

Aim:

iii)To demonstrate the use of a friend function to access private data in C++.

Description:

A friend function is not a member of the class but has the right to access its private and protected members.

It is declared inside the class using the keyword friend.

Program:

```
#include <iostream>
using namespace std;
class Box {
private:
    int length;
public:
    Box(int l) {
        length = l;
    }
    // Declare friend function
    friend void printLength(Box b);
};

// Friend function definition
void printLength(Box b) {
    cout << "Length of box = " << b.length << endl;
}
int main() {
    Box b1(15);
```

```
    printLength(b1); // accessing private data using friend function
    return 0;
}
```

Output:

Length of box = 15

Roll Number:

Practice-6

1)Aim:

To demonstrate the use of a default constructor and destructor in C++.

Description:

A constructor is a special function in a class that initializes objects.

A default constructor takes no parameters and is called automatically when an object is created.

A destructor is used to clean up resources and is called automatically when the object goes out of scope.

Program:

```
#include <iostream>
using namespace std;
class Demo {
public:
    Demo() {  // Default constructor
        cout << "Default Constructor Called!" << endl;
    }
     ~Demo() { // Destructor
        cout << "Destructor Called!" << endl;
    }
};
int main() {
    Demo obj; // Constructor will be called automatically
    cout << "Inside main function" << endl;
    return 0; // Destructor will be called automatically
}
```

Output:

Default Constructor Called!

Inside main function

Destructor Called!

2. Constructor Overloading

Aim:

To demonstrate constructor overloading in C++.

Description:

Constructor overloading means having multiple constructors with different parameter lists in the same class.

It helps in initializing objects in different ways.

Program:

```
#include <iostream>
using namespace std;
class Student {
private:
    int rollNo;
    string name;
public:
    Student() { // Default constructor
        rollNo = 0;
        name = "Unknown";
    }
        Student(int r) { // Constructor with one argument
        rollNo = r;
        name = "Not Provided";
    }
    Student(int r, string n) { // Constructor with two arguments
        rollNo = r;
        name = n;
    }
```

```
    void display() {
        cout << "Roll No: " << rollNo << ", Name: " << name << endl;
    }
};
int main() {
    Student s1;              // Calls default constructor
    Student s2(10);          // Calls constructor with one argument
    Student s3(20, "Alice"); // Calls constructor with two arguments
    s1.display();
    s2.display();
    s3.display();
    return 0;
}
```

Output:

Roll No: 0, Name: Unknown

Roll No: 10, Name: Not Provided

Roll No: 20, Name: Alice

3)Aim:

To demonstrate the use of a copy constructor in C++.

Description:

A copy constructor is used to initialize an object using another object of the same class.

If not defined, the compiler provides a default one (shallow copy).

We can also define our own copy constructor.

Program:

```
#include <iostream>
using namespace std;
class Book {
private:
    string title;
    int pages;
public:
    // Parameterized constructor
    Book(string t, int p) {
        title = t;
        pages = p;
    }
    // Copy constructor
    Book(const Book &b) {
        title = b.title;
        pages = b.pages;
    }
    void display() {
```

```
        cout << "Title: " << title << ", Pages: " << pages << endl;
    }
};

int main() {
    Book b1("C++ Programming", 350);  // Normal constructor
    Book b2 = b1;  // Copy constructor
    cout << "Original Book: ";
    b1.display();
    cout << "Copied Book: ";
    b2.display();
    return 0;
}
```

Output:

Original Book: Title: C++ Programming, Pages: 350

Copied Book: Title: C++ Programming, Pages: 350

Roll Number:

Practice-7

1. Single Inheritance

Aim:

i)To demonstrate single inheritance in C++.

Description:

In single inheritance, a derived class inherits from a single base class.

Program:

```
#include <iostream>
using namespace std;
class Animal {
public:
    void eat() {
        cout << "Animal is eating" << endl;
    }
};
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog is barking" << endl;
    }
};
int main() {
    Dog d;
    d.eat();  // inherited function
    d.bark(); // own function
    return 0;
}
```

Output:

Animal is eating

Dog is barking

2. Multiple Inheritance

Aim:

ii)To demonstrate multiple inheritance in C++.

Description:

In multiple inheritance, a derived class inherits from two or more base classes.

Program:

```
#include <iostream>
using namespace std;
class Teacher {
public:
    void teach() {
        cout << "Teaching students" << endl;
    }
};
class Researcher {
public:
    void research() {
        cout << "Conducting research" << endl;
    }
};
class Professor : public Teacher, public Researcher {
public:
    void guide() {
        cout << "Guiding students" << endl;
    }
```

```
};
int main() {
    Professor p;
    p.teach();
    p.research();
    p.guide();
    return 0;
}
```

Output:

Teaching students

Conducting research

Guiding students

3. Multilevel Inheritance

Aim:

iii)To demonstrate multilevel inheritance in C++.

Description:

In multilevel inheritance, a derived class becomes a base class for another class.

Program:

```
#include <iostream>
using namespace std;
class Grandparent {
public:
    void showGrandparent() {
        cout << "I am Grandparent" << endl;
    }
};
class Parent : public Grandparent {
public:
    void showParent() {
        cout << "I am Parent" << endl;
    }
};
class Child : public Parent {
public:
    void showChild() {
        cout << "I am Child" << endl;
    }
};
```

```
int main() {
    Child c;
    c.showGrandparent();
    c.showParent();
    c.showChild();
    return 0;
}
```

Output:

I am Grandparent

I am Parent

I am Child

4. Hierarchical Inheritance

Aim:

iv)To demonstrate hierarchical inheritance in C++.

Description:

In hierarchical inheritance, multiple classes inherit from the same base class.

Program:

```
#include <iostream>
using namespace std;
class Shape {
public:
    void display() {
        cout << "This is a shape" << endl;
    }
};
class Circle : public Shape {
public:
    void area() {
        cout << "Area of circle = πr²" << endl;
    }
};
class Rectangle : public Shape {
public:
    void area() {
        cout << "Area of rectangle = l × b" << endl;
    }
};
int main() {
```

```
    Circle c;
    Rectangle r;
    c.display();
    c.area();
    r.display();
    r.area();
    return 0;
}
```

Output:

This is a shape

Area of circle = πr²

This is a shape

Area of rectangle = l × b

5. Hybrid Inheritance

Aim:

v)To demonstrate hybrid inheritance in C++.

Description:

Hybrid inheritance is a combination of two or more types of inheritance (e.g., multiple + multilevel).

Here, a class inherits from two base classes, one of which is already derived from another class.

Program:

```
#include <iostream>
using namespace std;
class Person {
public:
    void showPerson() {
        cout << "I am a Person" << endl;
    }
};
class Employee : public Person {
public:
    void showEmployee() {
        cout << "I am an Employee" << endl;
    }
};
class Student {
public:
    void showStudent() {
        cout << "I am a Student" << endl;
    }
```

```
};
class WorkingStudent : public Employee, public Student {
public:
    void showWorkingStudent() {
        cout << "I am a Working Student" << endl;
    }
}
int main() {
    WorkingStudent ws;
    ws.showPerson();       // from Person via Employee
    ws.showEmployee();     // from Employee
    ws.showStudent();      // from Student
    ws.showWorkingStudent();
    return 0;
}
```


Output:

I am a Person

I am an Employee

I am a Student

I am a Working Student

Roll Number:

Practice-8

1)Aim: To demonstrate the order of execution of constructors and destructors in inheritance.

Description:

When a derived class object is created, constructors are called from base class to derived class (top-down order).

When the object goes out of scope, destructors are called in the reverse order (derived to base).

This chaining ensures proper initialization and cleanup of resources.

Program:

```
#include <iostream>
using namespace std;
class Base {
public:
    Base() {
        cout << "Base Constructor Called" << endl;
    }
    ~Base() {
        cout << "Base Destructor Called" << endl;
    }
};
class Derived : public Base {
public:
    Derived() {
        cout << "Derived Constructor Called" << endl;
    }
    ~Derived() {
        cout << "Derived Destructor Called" << endl;
    }
```

```
};
class Child : public Derived {
public:
    Child() {
        cout << "Child Constructor Called" << endl;
    }
    ~Child() {
        cout << "Child Destructor Called" << endl;
    }
};
int main() {
    cout << "Creating object..." << endl;
    Child obj;   // Constructors will be called
    cout << "Object created successfully!" << endl;
    cout << "Exiting main..." << endl;
    return 0;    // Destructors will be called automatically
}
```

Output:

Creating object...

Base Constructor Called

Derived Constructor Called

Child Constructor Called

Object created successfully!

Exiting main...

Child Destructor Called

Derived Destructor Called

Base Destructor Called

Roll Number:

Practice-9

1)Aim:

To demonstrate the order of execution of constructors and destructors in inheritance.

Description:

When a derived class object is created, constructors are called from base class to derived class (top-down order).

When the object goes out of scope, destructors are called in the reverse order (derived to base).

This chaining ensures proper initialization and cleanup of resources.

Program:

```
#include <iostream>
using namespace std;
class Base {
public:
    Base() {
        cout << "Base Constructor Called" << endl;
    }
    ~Base() {
        cout << "Base Destructor Called" << endl;
    }
};
class Derived : public Base {
public:
    Derived() {
        cout << "Derived Constructor Called" << endl;
    }
    ~Derived() {
        cout << "Derived Destructor Called" << endl;
```

```
    }
};
class Child : public Derived {
public:
    Child() {
        cout << "Child Constructor Called" << endl;
    }
    ~Child() {
        cout << "Child Destructor Called" << endl;
    }
};
int main() {
    cout << "Creating object..." << endl;
    Child obj;   // Constructors will be called
    cout << "Object created successfully!" << endl;
    cout << "Exiting main..." << endl;
    return 0;    // Destructors will be called automatically
}
```

Output:

Creating object...

Base Constructor Called

Derived Constructor Called

Child Constructor Called

Object created successfully!

Exiting main...

Child Destructor Called

Derived Destructor Called

Base Destructor Called

Roll Number:

Practice-10

1)Aim:

To demonstrate the use of pointers to objects and accessing class members in C++.

Description:

A class object can be accessed using a pointer with the -> operator.

This is commonly used for dynamic memory allocation (new) and polymorphism.

In this example, we create a pointer to a Student object and use it to access members.

Program:

```
#include <iostream>
using namespace std;
class Student {
private:
    int rollNo;
    string name;
public:
    void setData(int r, string n) {
        rollNo = r;
        name = n;
    }
    void display() {
        cout << "Roll No: " << rollNo << ", Name: " << name << endl;
    }
};
int main() {
    Student s1;         // Normal object
    Student *ptr = &s1;  // Pointer to object
```

```
    ptr->setData(101, "Alice");
    ptr->display();
    // Dynamic allocation
    Student *s2 = new Student;
    s2->setData(102, "Bob");
    s2->display();
    delete s2; // Free memory
    return 0;
}
```

Output:

Roll No: 101, Name: Alice

Roll No: 102, Name: Bob

2)Aim:

ii)To demonstrate the use of virtual base classes to resolve the diamond problem in C++.

Description:

The diamond problem occurs when a class inherits from two classes that have a common base, causing ambiguity

Virtual inheritance ensures only one copy of the base class is inherited, solving the problem.

Program:

```
#include <iostream>
using namespace std;
class Person {
public:
    void show() {
        cout << "I am a Person" << endl;
    }
};
class Student : virtual public Person {
public:
    void showStudent() {
        cout << "I am a Student" << endl;
    }
};
class Employee : virtual public Person {
public:
    void showEmployee() {
        cout << "I am an Employee" << endl;
    }
```

```
};

class WorkingStudent : public Student, public Employee {
public:
    void showWorkingStudent() {
        cout << "I am a Working Student" << endl;
    }
};
int main() {
    WorkingStudent ws;
    ws.show();              // No ambiguity due to virtual inheritance
    ws.showStudent();
    ws.showEmployee();
    ws.showWorkingStudent();
    return 0;
}
```

Roll Number:

Output:

I am a Person

I am a Student

I am an Employee

I am a Working Student

Roll Number:

Practice-11

Aim:

To demonstrate function overriding and dynamic dispatch using base class pointers and virtual functions.

Description:

A virtual function in C++ is a function declared in the base class with the keyword virtual, which allows it to be overridden in derived classes.

Runtime polymorphism (dynamic dispatch) occurs when a base class pointer or reference calls a function that is actually resolved at runtime depending on the type of the object it points to.

Without virtual, function calls are resolved at compile time (static binding). With virtual, they are resolved at runtime (dynamic binding).

Program:

```
#include <iostream>
using namespace std;
class Shape {
public:
    // Virtual function
    virtual void draw() {
        cout << "Drawing a generic shape" << endl;
    }
};
class Circle : public Shape {
public:
    void draw() override { // Function overriding
        cout << "Drawing a Circle" << endl;
    }
};
class Rectangle : public Shape {
```

```
public:
    void draw() override {
        cout << "Drawing a Rectangle" << endl;
    }
};
int main() {
    Shape *ptr;   // Base class pointer
    Circle c;
    Rectangle r;
    // Base pointer pointing to Circle
    ptr = &c;
    ptr->draw();   // Calls Circle's draw() (runtime polymorphism)
    // Base pointer pointing to Rectangle
    ptr = &r;
    ptr->draw();   // Calls Rectangle's draw() (runtime polymorphism)
    return 0;
}
```

Output:

Drawing a Circle

Drawing a Rectangle

Roll Number:

Practice-12

1) Function Template

Aim:

To demonstrate a function template for performing the same operation on different data types.

Description:

Templates allow writing generic code.

A function template enables a single function to work with multiple data types (int, float, etc.).

Program:

```
#include <iostream>
using namespace std;
// Function template
template <class T>
T add(T a, T b) {
    return a + b;
}
int main() {
    cout << "Sum of Integers: " << add(10, 20) << endl;
    cout << "Sum of Doubles: " << add(3.5, 2.7) << endl;
    cout << "Sum of Characters: " << add('A', (char)2) << endl; // 'A' + 2
    return 0;
}
```

Output:

Sum of Integers: 30

Sum of Doubles: 6.2

Sum of Characters: C

2) Class Template

Aim:

To demonstrate a class template for creating a generic class.

Description:

A class template defines a blueprint for creating classes that can work with any data type.

Program:

```
#include <iostream>
using namespace std;
// Class template
template <class T>
class Box {
private:
    T value;
public:
    void setValue(T v) { value = v; }
    T getValue() { return value; }
};
int main() {
    Box<int> intBox;
    Box<string> strBox;
    intBox.setValue(100);
    strBox.setValue("Hello Templates");
    cout << "Integer Value: " << intBox.getValue() << endl;
    cout << "String Value: " << strBox.getValue() << endl;
    return 0;
}
```

Output:

Integer Value: 100

String Value: Hello Templates

Roll Number:

Practice-13

1) Try, Catch, Throw

Aim:

To demonstrate exception handling using try, throw, and catch in C++.

Description:

Exceptions handle runtime errors gracefully.

A block of risky code is placed inside try.

If an error occurs, throw is used, and the exception is caught using catch.

Program:

```
#include <iostream>
using namespace std;
int main() {
    int a, b;
    cout << "Enter two numbers: ";
    cin >> a >> b;
    try {
        if (b == 0)
            throw "Division by Zero Error!";
        cout << "Result: " << (a / b) << endl;
    }
    catch (const char* msg) {
        cout << "Exception Caught: " << msg << endl;
    }
    return 0;
}
```

Roll Number:

Input:

Enter two numbers: 10 0

Output:

Exception Caught: Division by Zero Error!

2) Multiple Catch Blocks

Aim: To demonstrate multiple catch blocks for handling different exception types.

Program:

```
#include <iostream>
using namespace std;
int main() {
    try {
        int x;
        cout << "Enter a number: ";
        cin >> x;
        if (x == 0)
            throw x;
        else if (x < 0)
            throw string("Negative Number Exception");
        else
            cout << "You entered: " << x << endl;
    }
    catch (int n) {
        cout << "Exception: Division by Zero not allowed!" << endl;
    }
    catch (string &msg) {
        cout << "Exception: " << msg << endl;
    }
    return 0;
}
```

Roll Number:

Input:

Enter a number: -5

Output:

Exception: Negative Number Exception

Roll Number:

Practice-14

1) List and Vector

Aim:

To demonstrate insertion, deletion, and traversal using list and vector.

Program:

```
#include <iostream>
#include <list>
#include <vector>
using namespace std;
int main() {
    // Vector
    vector<int> v = {1, 2, 3};
    v.push_back(4);
    v.erase(v.begin()); // delete first element
    cout << "Vector Elements: ";
    for (int x : v) cout << x << " ";
    cout << endl;
    // List
    list<int> l = {10, 20, 30};
    l.push_front(5);
    l.push_back(40);
    l.remove(20); // delete specific element
    cout << "List Elements: ";
    for (int x : l) cout << x << " ";
    cout << endl;
    return 0;
}
```

Output:

Vector Elements: 2 3 4

List Elements: 5 10 30 40

2) Deque

Aim:

To demonstrate basic operations using deque (double-ended queue).

Program:

```
#include <iostream>
#include <deque>
using namespace std;
int main() {
    deque<int> dq;
    dq.push_back(10);
    dq.push_front(5);
    dq.push_back(15);
    dq.pop_front(); // removes first element
    cout << "Deque Elements: ";
    for (int x : dq) cout << x << " ";
    cout << endl;
    return 0;
}
```

Output:

Deque Elements: 10 15

3) Map

Aim:

To demonstrate insertion, deletion, access, and searching using map.

Program:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<int, string> m;
    // Insertion
    m[1] = "One";
    m[2] = "Two";
    m[3] = "Three";
    // Traversal
    cout << "Map Elements:" << endl;
    for (auto &p : m)
        cout << p.first << " -> " << p.second << endl;
    // Search
    if (m.find(2) != m.end())
        cout << "Found Key 2: " << m[2] << endl;
    // Deletion
    m.erase(1);
    cout << "After Deletion:" << endl;
    for (auto &p : m)
```

```
        cout << p.first << " -> " << p.second << endl;
    return 0;
}
```

Output:

Map Elements:

1 -> One

2 -> Two

3 -> Three

Found Key 2: Two

After Deletion:

2 -> Two

3 -> Three