UNIT 1

EXPERIMENT 1

AIM: To find the roots of a quadratic equation using C++.

Description:

To develop a C++ program that finds the roots of a quadratic equation of the form:
$ax^2 + bx + c = 0$

The program begins by prompting the user to input the coefficients a, b, and c. It then calculates the **discriminant ($D = b^2 - 4ac$)** to determine the nature of the roots:

- If D > 0, the equation has **two real and distinct roots**.
- If D == 0, the equation has **two real and equal roots**.
- If D < 0, the equation has **two complex (imaginary) roots**.

Procedure or Algorithm:

Depending on the value of the discriminant, the program uses the appropriate formula to compute and display the roots. It also provides meaningful messages to indicate the nature of the roots to the user. Use of **conditional statements, mathematical functions**, and **input/output operations** in C++ to develop the program.

Program:

```
// Find the roots of a quadratic equation (ax2 + bx + c)
// This code handles real and distinct, real and equal, and complex roots.

#include <iostream>
#include <cmath>                // for sqrt() function
using namespace std;

int main()
{
   float a, b, c;
   cout << "Enter coefficients a, b and c: ";
   cin >> a >> b >> c;

   float discriminant = b*b - 4*a*c;
   float root1, root2;

   if (a == 0) {
      cout << "This is not a quadratic equation (a cannot be 0)." << endl;
   }
   else if (discriminant > 0) {
      // Real and distinct roots
      root1 = (-b + sqrt(discriminant)) / (2*a);
```

```
        root2 = (-b - sqrt(discriminant)) / (2*a);
        cout << "Roots are real and distinct." << endl;
        cout << "Root 1 = " << root1 << endl;
        cout << "Root 2 = " << root2 << endl;
    }
    else if (discriminant == 0) {
        // Real and equal roots
        root1 = root2 = -b / (2*a);
        cout << "Roots are real and equal." << endl;
        cout << "Root = " << root1 << endl;
    }
    else {
        // Complex roots
        float realPart = -b / (2*a);
        float imagPart = sqrt(-discriminant) / (2*a);
        cout << "Roots are complex and imaginary." << endl;
        cout << "Root 1 = " << realPart << " + " << imagPart << "i" << endl;
        cout << "Root 2 = " << realPart << " - " << imagPart << "i" << endl;
    }

    return 0;
}
```

Output:

Sample Input/Output:
Enter coefficients a, b and c: 1 5 6
Roots are real and distinct.
Root 1 = -2
Root 2 = -3
Sample Input/Output:
Enter coefficients a, b and c: 1 4 4
Roots are real and equal.
Root = -2
Sample Input/Output:
Enter coefficients a, b and c: 1 2 5
Roots are complex and imaginary.
Root 1 = -1 + 2i
Root 2 = -1 - 2i

EXPERIMENT 2

AIM: Find factorial of a given number using recursion

Description:

The factorial of a non-negative integer n is the product of all positive integers less than or equal to n. It is denoted as:

n!=n×(n−1)×(n−2)×…×1n! = n \times (n-1) \times (n-2) \times \ldots \times 1n!=n×(n−1)×(n−2)×…×1

By definition:

- 0! = 1
- 1! = 1

Using recursion, the factorial function can be defined as:

Recursion is a programming technique where a function calls itself to solve smaller sub-problems of the original problem.

Algorithm:

Start

Read a positive integer n

If n < 0, display an error message and stop

Call the recursive function factorial(n)

In the recursive function:

If n is 0 or 1, return 1

Else return n * factorial(n-1)

Display the result

Stop

Program:

```
//Find factorial of a given number using recursion
#include <iostream>
using namespace std;
// Recursive function to calculate factorial
long factorial(int n)
{
```

```
    if (n == 0 || n==1)
        return 1;
    else
        return n * factorial(n - 1);
}
int main()
{
    int number;
    cout << "Enter a positive integer: ";
    cin >> number;
    if (number < 0)
          {       cout << "Factorial is not defined for negative numbers." << endl;
    }
          else
          {
        long result = factorial(number);
        cout << "Factorial of " << number << " = " << result << endl;
    }
    return 0;
}
```

OUPUT:

```
Enter a positive integer: 5
Factorial of 5 = 120
Enter a positive integer: -3
Factorial is not defined for negative numbers.
```

EXPERIMENT 3

AIM: Implement scope resolution and namespaces.

Description:

To understand and demonstrate the use of the **scope resolution operator (::)** and **namespace** feature in C++ for managing variable scope and avoiding name conflicts.

Scope Resolution Operator (::):

The scope resolution operator is used to:

- Access a **global variable** when a local variable with the same name exists.
- Define functions outside class definitions.
- Access static members of a class.

Namespace:

- Namespaces allow grouping of variables, functions, and classes to prevent **name conflicts**, especially in large projects or when using multiple libraries

Algorithm:

1. Define a global variable x
2. Create a custom namespace MyNamespace with a variable and a function
3. In main():
    - Define a local variable x
    - Print local x
    - Use ::x to print global x
    - Access and display namespace variable and function
4. End the program

Program

```
// Implement Scope Resolution Operation and namespace

#include <iostream>

using namespace std;  // using standard namespace

int x = 100;                    // Global variable

namespace MyNamespace

{
```

```
    int value = 50;
    void show()
        {       cout << "Inside MyNamespace, value = " << value << endl;
    }
}
int main()
{   int x = 10; // Local variable
    cout << "Local x = " << x << endl;
    cout << "Global x = " << ::x << endl; // Using scope resolution to access global x
    // Accessing namespace variable and function
    cout << "Namespace value = " << MyNamespace::value << endl;
    MyNamespace::show();
    return 0;
}
```

EXPERIMENT 4

Aim: Illustrate the use of default arguments and access specifiers

Description:

Default Arguments:

Default arguments allow a function to assign values automatically when corresponding arguments are missing during the function call. They reduce code redundancy.

Example:

void greet(string name = "Guest", int times = 1);

Access Specifiers:

Access specifiers control the visibility of class members:

- public: Accessible from anywhere
- private: Accessible only within the class
- protected: Accessible within the class and its derived classes

Procedure:

Define a function with default arguments.
Call the function in main() using:

- No arguments
- One argument
- Two arguments

Create a class with variables under all three access specifiers.
Access and display values **using public member function.**

Program:

```
#include <iostream>
using namespace std;
// Function with default arguments
void greet(string name = "Guest", int times = 1)
{
   for (int i = 0; i < times; i++) {
      cout << "Hello, " << name << "!" << endl;
   }
}

// Class to illustrate access specifiers
class DemoAccess {
private:
   int privateValue = 100;
```

```
protected:
    int protectedValue = 200;

public:
    int publicValue = 300;

    void showValues() {
        cout << "Private Value: " << privateValue << endl;
        cout << "Protected Value: " << protectedValue << endl;
        cout << "Public Value: " << publicValue << endl;
    }
};

int main()
{
    // Demonstrate default arguments
    greet();                // Uses both default arguments
    greet("Anil");          // Uses default for 'times'
    greet("Ravi", 3);       // No default used

    // Demonstrate access specifiers
    DemoAccess obj;
    obj.showValues();               // Access through public function
    cout << "Direct Access to Public Value: " << obj.publicValue << endl;

    // The following would cause errors:
    // cout << obj.privateValue;
    // cout << obj.protectedValue;

    return 0;
}
```

Sample Input/Output:

```
Hello, Guest!
Hello, Anil!
Hello, Ravi!
Hello, Ravi!
Hello, Ravi!
Private Value: 100
Protected Value: 200
Public Value: 300
Direct Access to Public Value: 300
```

UNIT – 2

EXPERIMENT 1

Aim: Program to illustrate inline functions and function overloading.

Description

1. **Inline Functions:**
    - An inline function is a function in which the compiler places a copy of the code directly at the point where the function is called, instead of performing a normal function call.
    - It reduces the overhead of a function call and improves execution speed for small, frequently used functions.
2. **Function Overloading:**
    - Function overloading allows multiple functions with the same name but different parameter lists (number or type of arguments).
    - It increases program readability and reusability by using the same function name for different purposes.

Procedure

- Start the C++ compiler and create a new program file.
- Include the necessary header file <iostream>.
- Define an inline function to perform a simple operation (e.g., square of a number).
- Define multiple overloaded functions with the same name but different parameter lists.
- In the main() function, call the inline function and the overloaded functions to demonstrate their functionality.
- Compile and execute the program.
- Observe the output and verify correctness.

Program

```
#include <iostream>

using namespace std;

// Inline function to calculate square of a number

inline int square(int x) {

   return x * x;

}

// Function Overloading examples
```

```
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() {
    cout << "Inline Function Example:" << endl;
    cout << "Square of 5 = " << square(5) << endl;
    cout << "Square of 10 = " << square(10) << endl;

    cout << "\nFunction Overloading Example:" << endl;
    cout << "Addition of 2 and 3 = " << add(2, 3) << endl;
    cout << "Addition of 2.5 and 3.7 = " << add(2.5, 3.7) << endl;
    cout << "Addition of 1, 2 and 3 = " << add(1, 2, 3) << endl;

    return 0;
}
```

Sample Output

```
Inline Function Example:
Square of 5 = 25
Square of 10 = 100
Function Overloading Example:
Addition of 2 and 3 = 5
Addition of 2.5 and 3.7 = 6.2
Addition of 1, 2 and 3 = 6
```

EXPERIMENT 2

AIM: Progam to illustrate friend function

Description

- A friend function in C++ is a special function that is not a member of a class, but it can access the private and protected members of that class.
- It is declared inside the class with the keyword friend.
- Friend functions are commonly used when two or more classes need to share information or when operator overloading is required.

Procedure

1. Create a class with private data members.
2. Declare a function as **friend** inside the class.
3. Define the friend function outside the class scope.
4. In the friend function, access and manipulate private members of the class.
5. In the main() function, create an object of the class and call the friend function.
6. Compile and execute the program to verify results.

Program

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

class Box {
private:
   int length;
public:
   Box(int l) {
      length = l;
   }
   // Declaration of friend function
   friend void showLength(Box b);
};
```

```
// Definition of friend function
void showLength(Box b) {
    cout << "Length of the box = " << b.length << endl;
}

int main() {
    Box b1(25);
    showLength(b1);  // Friend function accessing private member
    return 0;
}
```

Sample Output

```
Length of the box = 25
```

EXPERIMENT 3

AIM: Program to illustrate the use of Constructors and Destructors.

Description

1. **Constructor**
    - A constructor is a special member function of a class that is automatically invoked when an object of the class is created.
    - It has the same name as the class and does not have a return type.
    - Constructors are mainly used to initialize data members of a class.
2. **Destructor**
    - A destructor is a special member function of a class that is automatically invoked when an object goes out of scope or is explicitly destroyed.
    - It has the same name as the class prefixed with a tilde (~).
    - It is mainly used to release resources such as memory or file handles.

Procedure

1. Define a class with data members.
2. Write a **constructor** to initialize data members.
3. Write a **destructor** to display a message when the object is destroyed.
4. In the main() function, create objects of the class.
5. Observe how constructors are invoked during object creation and destructors during object destruction.
6. Compile and execute the program.

Program

```
#include <iostream>

using namespace std;

class Student {

private:

   string name;

   int age;

public:

   // Constructor
```

```
    Student(string n, int a) {
        name = n;
        age = a;
        cout << "Constructor called for " << name << endl;
    }
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
    // Destructor
    ~Student() {
        cout << "Destructor called for " << name << endl;
    }
};
int main() {
    Student s1("Rahul", 20);
    s1.display();
    {
        Student s2("Priya", 19);
        s2.display();
    } // Destructor for s2 is called here (end of scope)
    return 0;
}
```

Sample Output:

```
Constructor called for Rahul
Name: Rahul, Age: 20
Constructor called for Priya
Name: Priya, Age: 19
Destructor called for Priya
Destructor called for Rahul
```

EXPERIMENT 4

AIM: Program illustrating Constructor overloading.

Description

- **Constructor Overloading**:
  In C++, more than one constructor can be defined in a class with different parameter lists.
  This is called **constructor overloading** and it allows objects to be initialized in different ways.
- Example:
  - A default constructor initializes data with fixed values.
  - A parameterized constructor initializes data with values passed by the user.
  - Another constructor can initialize only selected attributes.

Procedure

1. Create a class with data members.
2. Define multiple constructors with different parameter lists (default, parameterized, copy, etc.).
3. Use constructor overloading to initialize objects differently.
4. In the main() function, create objects using different constructors.
5. Compile and run the program to observe how different constructors are invoked.

Program

```
#include <iostream>

using namespace std;

class Student {

private:

   string name;

   int age;

public:

   // Default Constructor

   Student() {

      name = "Unknown";

      age = 0;
```

```
        cout << "Default Constructor called" << endl;
    }
    // Parameterized Constructor
    Student(string n, int a) {
        name = n;
        age = a;
        cout << "Parameterized Constructor called" << endl;
    }
    // Constructor with only one parameter
    Student(string n) {
        name = n;
        age = -1;
        cout << "Single Parameter Constructor called" << endl;
    }
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;    }
};
int main() {
    // Default constructor
    Student s1;
    s1.display();
    // Parameterized constructor
    Student s2("Rahul", 20);
    s2.display();
    // Constructor with single parameter
    Student s3("Priya");
    s3.display();
    return 0;
}
```

Sample Output

Default Constructor called

Name: Unknown, Age: 0

Parameterized Constructor called

Name: Rahul, Age: 20

Single Parameter Constructor called

Name: Priya, Age: -1

EXPERIMENT 5

AIM: Program illustrating Copy Constructor.

Description

- A **copy constructor** is a special type of constructor used to create a new object as a copy of an existing object.
- Syntax:

```
ClassName(const ClassName &obj) {
  // copy code here                }
```

It is invoked when:

1. A new object is initialized with an existing object.
2. An object is passed by value to a function.
3. An object is returned from a function by value.

Procedure

1. Define a class with private data members.
2. Write a **parameterized constructor** to initialize values.
3. Define a **copy constructor** that takes a reference to an object of the same class.
4. In the main() function, create one object using the parameterized constructor and another using the copy constructor.
5. Display values to confirm that the data is copied correctly.
6. Compile and execute the program.

Program Code:

```
#include <iostream>
using namespace std;
class Student {
private:
   string name;
   int age;
public:
   // Parameterized Constructor
   Student(string n, int a) {
```

```
        name = n;
        age = a;
        cout << "Parameterized Constructor called" << endl;
    }
    // Copy Constructor
    Student(const Student &s) {
        name = s.name;
        age = s.age;
        cout << "Copy Constructor called" << endl;
    }
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
int main() {
    // Creating object using parameterized constructor
    Student s1("Rahul", 20);
    s1.display();
    // Creating object using copy constructor
    Student s2(s1);   // Copy s1 into s2
    s2.display();
    return 0;
}
```

Sample Output:

Parameterized Constructor called

Name: Rahul, Age: 20

Copy Constructor called

Name: Rahul, Age: 20

UNIT 3

EXPERIMENT 1

Aim : Program to overload Unary and Binary Operators using Member Functions in C++.

Description

- **Operator Overloading** allows giving special meaning to operators when applied to user-defined types (objects).
- **Unary Operator Overloading**:
  - Works on a single operand (e.g., ++, --, -, !).
  - Example: Overloading ++ to increment object data.
- **Binary Operator Overloading**:
  - Works on two operands (e.g., +, -, *, /).
  - Example: Overloading + to add two objects.

Procedure

1. Create a class with integer data members.
2. Define a **unary operator function** (e.g., operator++()) as a member function.
3. Define a **binary operator function** (e.g., operator+()) as a member function.
4. In main(), create objects and demonstrate operator overloading.
5. Compile and execute the program to verify correctness.

Program

```
#include <iostream>
using namespace std;
class Number {
private:
   int value;
public:
   Number(int v = 0) {
      value = v;
   }
   void operator++() {
```

```
        ++value;
    }
    Number operator+(Number obj) {
        Number temp;
        temp.value = value + obj.value;
        return temp;
    }
    void display() { cout << "Value: " << value << endl;
    }
};
int main() {
    Number n1(5), n2(10), n3;
    cout << "Before Unary Operation:" << endl;
    n1.display();
    ++n1;
    cout << "After Unary Operation (++n1):" << endl;
    n1.display();
    n3 = n1 + n2;
    cout << "After Binary Operation (n1 + n2):" << endl;
    n3.display();
    return 0;
}
```

Sample Output

Before Unary Operation:

Value: 5

After Unary Operation (++n1):

Value: 6

After Binary Operation (n1 + n2):

Value: 16

EXPERIMENT 2

Aim: Program to **overload Unary and Binary Operators using Friend Functions** in C++.

Description

- **Operator Overloading with Friend Functions** allows redefining operators while keeping the operator function **outside the class**, but still having access to private members using the friend keyword.
- **Unary Operator Overloading**:
  - Works on a single operand (e.g., ++, --).
  - Example: Overloading ++ to increment object data.
- **Binary Operator Overloading**:
  - Works on two operands (e.g., +, -).
  - Example: Overloading + to add two objects.

Procedure

1. Create a class with private data members.
2. Declare **friend functions** inside the class for operator overloading.
3. Define the operator functions outside the class scope.
4. In main(), create objects and demonstrate unary and binary operations.
5. Compile and execute the program to verify results.

Program

```
#include <iostream>
using namespace std;
class Number {
private:
   int value;
public:
   // Constructor
   Number(int v = 0) {
      value = v;
   }
```

```
    // Friend function declarations
    friend Number operator++(Number &n);      // Unary
    friend Number operator+(Number n1, Number n2);  // Binary
    void display() {
        cout << "Value: " << value << endl;
    }
};
// Unary operator overloading (prefix ++)
Number operator++(Number &n) {
    n.value++;
    return n;
}
// Binary operator overloading (+)
Number operator+(Number n1, Number n2) {
    return Number(n1.value + n2.value);
}
int main() {
    Number n1(5), n2(10), n3;
    cout << "Before Unary Operation:" << endl;
    n1.display();
    // Unary operator overloading
    ++n1;
    cout << "After Unary Operation (++n1):" << endl;
    n1.display();
    // Binary operator overloading
    n3 = n1 + n2;
    cout << "After Binary Operation (n1 + n2):" << endl;
    n3.display();
    return 0;
```

}

Sample output:

Before Unary Operation:

Value: 5

After Unary Operation (++n1):

Value: 6

After Binary Operation (n1 + n2):

Value: 16

EXPERIMENT 3

Aim

To write programs that demonstrate the different forms of inheritance in C++:

- Single Inheritance
- Multiple Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Description

Inheritance is one of the fundamental features of Object-Oriented Programming (OOP). It allows one class (derived class) to acquire the properties and behaviors of another class (base class). This experiment demonstrates all major forms of inheritance in C++.

Procedure

1. Define appropriate base classes with data members and member functions.
2. Create derived classes using different forms of inheritance.
3. Access base class properties in derived classes using constructors and member functions.
4. Compile and run the program to observe the behavior of each inheritance type.

Program

```
#include <iostream>

using namespace std;


// ---------------- Single Inheritance ----------------

class A {

public:

   void displayA() {

      cout << "This is class A (Base Class - Single Inheritance)" << endl;

   }

};
```

```
class B : public A {
public:
    void displayB() {
        cout << "This is class B (Derived from A - Single Inheritance)" << endl;
    }
};

// ---------- Multiple Inheritance ----------
class X {
public:
    void displayX() {
        cout << "This is class X (Base Class for Multiple Inheritance)" << endl;
    }
};

class Y {
public:
    void displayY() {
        cout << "This is class Y (Base Class for Multiple Inheritance)" << endl;
    }
};

class Z : public X, public Y {
public:
    void displayZ() {
        cout << "This is class Z (Derived from X and Y - Multiple Inheritance)" << endl;
    }
};
```

```
// ---------- Multi-level Inheritance ----------
class Parent {
public:
    void displayParent() {
        cout << "This is Parent Class (Multi-level Inheritance)" << endl;
    }
};

class Child : public Parent {
public:
    void displayChild() {
        cout << "This is Child Class (Derived from Parent)" << endl;
    }
};

class GrandChild : public Child {
public:
    void displayGrandChild() {
        cout << "This is GrandChild Class (Derived from Child - Multi-level)" << endl;
    }
};

// ---------- Hierarchical Inheritance ----------
class Base {
public:
    void displayBase() {
        cout << "This is Base Class (Hierarchical Inheritance)" << endl;
    }
```

```
};

class Derived1 : public Base {
public:
    void displayDerived1() {
        cout << "This is Derived1 (from Base)" << endl;
    }
};

class Derived2 : public Base {
public:
    void displayDerived2() {
        cout << "This is Derived2 (from Base)" << endl;
    }
};

// ----------------- Hybrid Inheritance -----------------
class M {
public:
    void displayM() {
        cout << "This is class M (Base of Hybrid Inheritance)" << endl;
    }
};

class N : public M {
public:
    void displayN() {
        cout << "This is class N (Derived from M)" << endl;
    }
```

```
};

class O {
public:
    void displayO() {
        cout << "This is class O (Independent Base for Hybrid Inheritance)" << endl;
    }
};

class P : public N, public O {
public:
    void displayP() {
        cout << "This is class P (Derived from both N and O - Hybrid)" << endl;
    }
};

// ----------------- Main Function -----------------
int main() {
    cout << "\n--- Single Inheritance ---" << endl;
    B;
    b.displayA();
    b.displayB();

    cout << "\n--- Multiple Inheritance ---" << endl;
    Z;
    z.displayX();
    z.displayY();
    z.displayZ();
```

```
    cout << "\n--- Multi-level Inheritance ---" << endl;
    GrandChild gc;
    gc.displayParent();
    gc.displayChild();
    gc.displayGrandChild();

    cout << "\n--- Hierarchical Inheritance ---" << endl;
    Derived1 d1;
    Derived2 d2;
    d1.displayBase();
    d1.displayDerived1();
    d2.displayBase();
    d2.displayDerived2();

    cout << "\n--- Hybrid Inheritance ---" << endl;
    P;
    p.displayM();
    p.displayN();
    p.displayO();
    p.displayP();

    return 0;
}
```

Sample Output

```
--- Single Inheritance ---
This is class A (Base Class - Single Inheritance)
This is class B (Derived from A - Single Inheritance)

--- Multiple Inheritance ---
```

This is class X (Base Class for Multiple Inheritance)

This is class Y (Base Class for Multiple Inheritance)

This is class Z (Derived from X and Y - Multiple Inheritance)

--- Multi-level Inheritance ---

This is Parent Class (Multi-level Inheritance)

This is Child Class (Derived from Parent)

This is GrandChild Class (Derived from Child - Multi-level)

--- Hierarchical Inheritance ---

This is Base Class (Hierarchical Inheritance)

This is Derived1 (from Base)

This is Base Class (Hierarchical Inheritance)

This is Derived2 (from Base)

--- Hybrid Inheritance ---

This is class M (Base of Hybrid Inheritance)

This is class N (Derived from M)

This is class O (Independent Base for Hybrid Inheritance)

This is class P (Derived from both N and O - Hybrid)

EXPERIMENT 4

Aim:

Program to illustrate the order of execution of constructors and destructors in inheritance in C++.

Description:

In C++, when classes are derived using inheritance, constructors and destructors are executed in a specific order:

- Constructors: Execution starts from the base class constructor and proceeds down the inheritance chain to the most derived class.
- Destructors: Execution happens in the reverse order, i.e., from the most derived class back to the base class.

This ensures that objects are properly initialized before they are used, and resources are released in the reverse order to avoid issues.

Procedure:

1. Create a base class with a constructor and destructor that display messages when called.
2. Create a derived class (single or multiple levels) with its own constructor and destructor.
3. Instantiate the derived class object and observe the order of constructor and destructor calls.
4. Extend the program to include multi-level inheritance for better observation.
5. Run and record the output.

Program

```
#include <iostream>

using namespace std;


// Base class

class Base {

public:

   Base() {
```

```
        cout << "Base class constructor called" << endl;
    }
    ~Base() {
        cout << "Base class destructor called" << endl;
    }
};
// Intermediate Derived class
class Derived1 : public Base {
public:
    Derived1() {
        cout << "Derived1 class constructor called" << endl;
    }
    ~Derived1() {
        cout << "Derived1 class destructor called" << endl;
    }     };
// Further Derived class
class Derived2 : public Derived1 {
public:
    Derived2() {
        cout << "Derived2 class constructor called" << endl;
    }
    ~Derived2() {
        cout << "Derived2 class destructor called" << endl;
    }     };

int main() {
    cout << "Creating object of Derived2 class..." << endl;
    Derived2 obj; // Object creation
```

```
    cout << "Exiting main..." << endl;
    return 0;
}
```

Sample Output

```
Creating object of Derived2 class...
Base class constructor called
Derived1 class constructor called
Derived2 class constructor called
Exiting main...
Derived2 class destructor called
Derived1 class destructor called
Base class destructor called
```

EXPERIMENT 5

Aim

To write C++ programs demonstrating:

1. Object as a class member
2. Pointer to a class
3. this pointer
4. Virtual Base Class

Description

- **Object as a class member:** When an object of one class is a data member of another class. This is known as **composition**.
- **Pointer to a class:** We can create pointers to objects of a class and access their members using the arrow (->) operator.
- **this pointer:** Every object in C++ has access to its own address through a special pointer named this. It is used to resolve ambiguity between data members and function parameters.
- **Virtual Base Class:** Used in multiple inheritance to avoid **diamond problem** (duplicate copies of base class members).

Procedure

1. Define separate programs for each concept.
2. Demonstrate **object as member** by including one class object inside another class.
3. Use **pointer to class** to access class members dynamically.
4. Demonstrate **this pointer** inside a member function to access object data members.
5. Use **virtual base class** to remove ambiguity in multiple inheritance.
6. Compile and run each program, observe the output.

Program

1. Object as a Class Member

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

```
class Address {
    string city;
public:
    Address(string c) { city = c; }
    void show() { cout << "City: " << city << endl; }
};

class Student {
    string name;
    Address addr;   // Object as a class member
public:
    Student(string n, string c) : name(n), addr(c) {}
    void display() {
        cout << "Name: " << name << endl;
        addr.show();
    }
};

int main() {
    Student s("Ravi", "Hyderabad");
    s.display();
    return 0;
}
```

2. Pointer to a Class

```
#include <iostream>
using namespace std;
class Box {
    int length;
```

```
public:
    void setLength(int l) { length = l; }
    void showLength() { cout << "Length: " << length << endl; }
};
int main() {
    Box b1;
    Box *ptr = &b1;   // Pointer to object
    ptr->setLength(15);
    ptr->showLength();
    return 0;
}
```

3. this Pointer

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

class Test {
    int x;
public:
    Test(int x) {
        this->x = x; // Resolving ambiguity
    }
    void show() { cout << "Value of x: " << this->x << endl; }
};

int main() {
    Test t(50);
    t.show();
    return 0;
}
```

4. Virtual Base Class

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

class Person {
public:
    void show() { cout << "I am a Person" << endl; }
};

class Student : virtual public Person {
public:
    void studentInfo() { cout << "I am a Student" << endl; }
};

class Teacher : virtual public Person {
public:
    void teacherInfo() { cout << "I am a Teacher" << endl; }
};
class TA : public Student, public Teacher {
public:
    void display() {
        show(); // Only one copy of Person is inherited
    }
};
int main() {
    TA obj;
    obj.display();
    obj.studentInfo();
    obj.teacherInfo();
    return 0;
}
```

EXPERIMENT 6

Aim:

To write a C++ program that demonstrates the use of **virtual functions** for achieving runtime polymorphism.

Description:

Virtual functions in C++ allow achieving **runtime polymorphism**.

- When a base class declares a function as virtual, it can be overridden in the derived class.
- A base class pointer (or reference) can call the derived class version of the function at runtime, enabling **dynamic dispatch**.
- Without virtual, the function call is resolved at compile time (**early binding**).
- With virtual, the function call is resolved at runtime (**late binding**).

This feature is widely used in **object-oriented programming** for implementing **polymorphism** and is essential in real-world applications such as GUI systems, game engines, and frameworks.

Procedure:

1. Create a base class with a virtual function.
2. Derive one or more classes from the base class and override the virtual function.
3. Use a base class pointer to refer to derived class objects.
4. Call the function through the base class pointer and observe the effect of virtual function (runtime binding).
5. Compile and execute the program to check outputs.

Program

```
#include <iostream>

using namespace std;

// Base Class

class Base {

public:

   virtual void display() {  // Virtual Function

      cout << "Display from Base class" << endl;

   }

};
```

```
// Derived Class 1
class Derived1 : public Base {
public:
    void display() override {
        cout << "Display from Derived1 class" << endl;
    }
};

// Derived Class 2
class Derived2 : public Base {
public:
    void display() override {
        cout << "Display from Derived2 class" << endl;
    }
};
int main() {
    Base* basePtr;  // Base class pointer
    Base b;
    Derived1 d1;
    Derived2 d2;

    // Base pointer pointing to Base object
    basePtr = &b;
    basePtr->display();
    // Base pointer pointing to Derived1 object
    basePtr = &d1;
    basePtr->display();

    // Base pointer pointing to Derived2 object
```

```
    basePtr = &d2;
    basePtr->display();

    return 0;
}
```

Expected Output:

Display from Base class

Display from Derived1 class

Display from Derived2 class

EXPERIMENT 7

Aim:

To write a C++ program that demonstrates the use of abstract classes and pure virtual functions to calculate the area of different shapes.

Description:

In C++, an **abstract class** is a class that cannot be instantiated and is designed to act as a base class. A class becomes abstract when it contains at least one **pure virtual function**.
A **pure virtual function** is declared by assigning = 0 in its declaration. Derived classes must provide definitions for pure virtual functions; otherwise, they too become abstract.

This experiment illustrates:

- How abstract classes enforce common functionality (e.g., area()) across multiple derived classes.
- How polymorphism allows working with different derived class objects using base class pointers.

Procedure:

1. Define an abstract base class Shape with a pure virtual function area().
2. Derive specific shape classes like Circle, Rectangle, and Triangle from Shape.
3. Implement the area() function in each derived class according to the shape formula.
4. Use base class pointer(s) to call area() polymorphically.
5. Display the calculated areas for different shapes.

Program:

```
#include <iostream>

#include <cmath>

using namespace std;

// Abstract Base Class

class Shape {

public:    // Pure virtual function

   virtual void area() = 0;    };

// Derived Class: Circle

class Circle : public Shape {
```

```
    float radius;
public:    Circle(float r) : radius(r) {}
                    void area() {
                    cout << "Area of Circle = " << 3.14159 * radius * radius << endl;
            }       };
// Derived Class: Rectangle
class Rectangle : public Shape {
    float length, breadth;
public:              Rectangle(float l, float b) : length(l), breadth(b) {}
                    void area() {
                     cout << "Area of Rectangle = " << length * breadth << endl;
                       }     };
// Derived Class: Triangle
class Triangle : public Shape {
    float base, height;
public:          Triangle(float b, float h) : base(b), height(h) {}
           void area() {
           cout << "Area of Triangle = " << 0.5 * base * height << endl;
            }       };
int main() {
    Shape *s; // Base class pointer
    Circle c(5);
    Rectangle r(4, 6);
    Triangle t(3, 7);
    s = &c;
    s->area();
    s = &r;
    s->area();
    s = &t;
    s->area();
```

```
    return 0;
}
```

Sample Output:

Area of Circle = 78.5398

Area of Rectangle = 24

Area of Triangle = 10.5

UNIT 4

EXPERIMENT 1

Aim: **Program to illustrate the use of function templates in C++.**

Description:

Templates in C++ allow writing generic programs that work with any data type. Function templates are particularly useful when the same logic applies to different data types (e.g., int, float, double, char). Instead of writing multiple overloaded functions, a single function template can be created. This promotes code reusability and type independence.

Example: Writing a template function for finding the maximum of two values.

Procedure:

1. Start the program by including the required header file.
2. Define a function template using the template keyword.
3. Write the generic function logic using placeholder type T.
4. Call the template function with different data types (int, float, char, etc.).
5. Compile and execute the program to observe the output.

Program:

```
#include <iostream>
using namespace std;
// Function template to find maximum of two values
template <class T>
T getMax(T a, T b) {
   return (a > b) ? a : b;
}
int main() {
   cout << "Maximum of 10 and 20 is: " << getMax(10, 20) << endl;
   cout << "Maximum of 5.5 and 2.3 is: " << getMax(5.5, 2.3) << endl;
   cout << "Maximum of 'A' and 'Z' is: " << getMax('A', 'Z') << endl;
   return 0;
}
```

Sample Output:

```
Maximum of 10 and 20 is: 20
```

Maximum of 5.5 and 2.3 is: 5.5

Maximum of 'A' and 'Z' is: Z

EXPERIMENT 2

Aim: Program to illustrate the implementation of a template class in C++.

Description:

Templates in C++ allow writing generic and reusable code. A template class enables defining a class with a placeholder for the data type, so the same class can handle different types of data without rewriting the code.

For example, a **template class** can be created for a stack or a simple operation like addition, and it can work with int, float, or double.

Procedure:

1. Start with the template <class T> keyword to define a generic class.
2. Create class members and functions using the generic type T.
3. Instantiate the class with different data types (e.g., int, float, double).
4. Perform operations and display results.
5. Compile and run the program to test with different types.

Program:

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

// Template Class
template <class T>
class Calculator {
   T num1, num2;
public:
   Calculator(T n1, T n2) {
      num1 = n1;
      num2 = n2;
   }

   void displayResult() {
      cout << "Numbers: " << num1 << " and " << num2 << endl;
      cout << "Addition: " << num1 + num2 << endl;
```

```
        cout << "Subtraction: " << num1 - num2 << endl;
        cout << "Multiplication: " << num1 * num2 << endl;
        cout << "Division: " << num1 / num2 << endl;
        cout << "-------------------------" << endl;
    }
};

int main() {
    // Integer type
    Calculator<int> intCalc(10, 5);
    cout << "Integer results:" << endl;
    intCalc.displayResult();

    // Float type
    Calculator<float> floatCalc(10.5, 5.2);
    cout << "Float results:" << endl;
    floatCalc.displayResult();

    // Double type
    Calculator<double> doubleCalc(20.5, 4.5);
    cout << "Double results:" << endl;
    doubleCalc.displayResult();

    return 0;
}
```

Sample Output:

```
Integer results:
Numbers: 10 and 5
Addition: 15
Subtraction: 5
```

Multiplication: 50

Division: 2

Float results:

Numbers: 10.5 and 5.2

Addition: 15.7

Subtraction: 5.3

Multiplication: 54.6

Division: 2.01923

Double results:

Numbers: 20.5 and 4.5

Addition: 25

Subtraction: 16

Multiplication: 92.25

Division: 4.55556

EXPERIMENT 3

Aim:

To implement and demonstrate class templates with multiple parameters in C++

Description:

Templates in C++ allow writing generic programs. A class template with multiple parameters enables us to define a class that can accept more than one data type at a time. This provides flexibility and reusability, as the same class can handle different data type combinations without rewriting code.

For example, a class can be designed to store and display values of two different types like int, float or string, char.

Procedure:

1. Start the C++ program by including the necessary header files.
2. Define a **class template** with two parameters (e.g., T1 and T2).
3. Create member functions to accept and display values of these parameters.
4. Instantiate objects of the class with different type combinations.
5. Compile and execute the program to verify the functionality.

Program:

```
#include <iostream>
using namespace std;
// Class template with two parameters
template <class T1, class T2>
class MyClass {
   T1 value1;
   T2 value2;
public:
   MyClass(T1 v1, T2 v2)     // Constructor
        {               value1 = v1;     value2 = v2;                    }
        void display()                    // Function to display values
        {             cout << "Value 1: " << value1 << ", Value 2: " << value2 << endl;
          }       };
int main() {
```

```
    MyClass<int, float> obj1(10, 20.5);         // Integer and float
    obj1.display();
    MyClass<string, char> obj2("Hello", 'A');          // String and char
    obj2.display();
    MyClass<double, int> obj3(99.99, 100);             // Double and int
    obj3.display();
    return 0;     }
```

Sample Output:

Value 1: 10, Value 2: 20.5

Value 1: Hello, Value 2: A

Value 1: 99.99, Value 2: 100

UNIT 5

EXPERIMENT 1

Aim:

To implement exception handling in C++ using **try, catch, and throw** for handling runtime errors.

Description:

Exception handling in C++ provides a mechanism to detect and handle runtime errors gracefully without crashing the program. It uses the keywords **try**, **catch**, and **throw**:

- **try block**: contains the code that might generate an exception.
- **throw**: signals the occurrence of an exception.
- **catch block**: handles the exception thrown by the program.

Common use cases include handling division by zero, invalid inputs, file errors, etc.

Procedure:

1. Start the C++ program and include the required headers.
2. Define a function or code block that may generate runtime errors.
3. Place the error-prone code inside a **try block**.
4. Use the **throw statement** to indicate an exception when an error occurs.
5. Provide one or more **catch blocks** to handle the exceptions.
6. Compile and run the program.
7. Test with different inputs to observe exception handling.

Program

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

int main() {
   int numerator, denominator;
   cout << "Enter numerator: ";
   cin >> numerator;
   cout << "Enter denominator: ";
   cin >> denominator;
      try {              if (denominator == 0) {
```

```
            throw runtime_error("Division by zero is not allowed!");
        }
        double result = (double) numerator / denominator;
        cout << "Result = " << result << endl;
    }
    catch (runtime_error &e) {
        cout << "Exception caught: " << e.what() << endl;
    }
    cout << "Program continues after exception handling..." << endl;
    return 0;
}
```

Sample Output:

```
Enter numerator: 10
Enter denominator: 0
Exception caught: Division by zero is not allowed!
Program continues after exception handling...
```

EXPERIMENT 2

Aim

To write a C++ program that demonstrates the use of multiple catch statements for handling different types of exceptions.

Description

In C++, exception handling is achieved using try, catch, and throw blocks. Sometimes, a program may throw different types of exceptions. To handle each type appropriately, multiple catch statements can be used. The compiler matches the exception type with the appropriate catch block. If no matching catch block is found, the program terminates abnormally.

This experiment helps to understand how multiple catch statements are used to handle different exception types in C++.

Procedure

1. Start the C++ program by including the necessary header file <iostream>.
2. Define a try block and write code that may throw different types of exceptions (e.g., integer, float, string).
3. Use multiple catch blocks, each designed to handle a specific type of exception.
4. Test the program by throwing different exceptions.
5. Compile and execute the program to verify correct exception handling.

Program

```
#include <iostream>

using namespace std;


int main() {
   try {
      int choice;
      cout << "Enter 1 for int exception, 2 for float exception, 3 for string exception: ";
      cin >> choice;


      if (choice == 1)
```

```
            throw 10;         // Integer exception
        else if (choice == 2)
            throw 3.14f;      // Float exception
        else if (choice == 3)
            throw "Hello";    // String exception
        else
            throw 'X';        // Character exception
    }
    catch (int e) {
        cout << "Caught an integer exception: " << e << endl;
    }
    catch (float e) {
        cout << "Caught a float exception: " << e << endl;
    }
    catch (const char* e) {
        cout << "Caught a string exception: " << e << endl;
    }
    catch (...) {
        cout << "Caught an unknown exception!" << endl;
    }

    cout << "Program continues after exception handling." << endl;
    return 0;
}
```

Sample Output

Enter 1 for int exception, 2 for float exception, 3 for string exception: 2

Caught a float exception: 3.14

Program continues after exception handling.

EXPERIMENT 3

Aim

Program to implement **List, Vector** and perform basic operations using the **STL (Standard Template Library)** in C++.

Description

The Standard Template Library (STL) in C++ provides useful classes like **vector** and **list** to store collections of elements.

- **Vector**: A dynamic array that allows random access and resizes automatically when elements are inserted/removed.
- **List**: A doubly linked list that allows fast insertion and deletion at any position but does not support random access.

Operations include:

- Insertion
- Deletion
- Traversal
- Size and capacity checking

This experiment demonstrates the usage of STL containers to handle collections efficiently.

Procedure

1. Include the header file <iostream> and <vector>, <list>.
2. Declare a vector and list of integers.
3. Perform insertion using push_back() or push_front() (for list).
4. Delete elements using pop_back() or remove().
5. Traverse elements using iterators.
6. Display the size and capacity of vector/list.
7. Execute and verify the output.

Program

```
#include <iostream>

#include <vector>
```

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

int main() {
    // Vector operations
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);

    cout << "Vector elements: ";
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << "\nSize: " << v.size() << ", Capacity: " << v.capacity() << endl;

    v.pop_back();
    cout << "After pop_back, Vector elements: ";
    for (auto it = v.begin(); it != v.end(); ++it)
        cout << *it << " ";
    cout << endl;

    // List operations
    list<int> l;
    l.push_back(100);
    l.push_front(50);
    l.push_back(150);

    cout << "\nList elements: ";
    for (auto it = l.begin(); it != l.end(); ++it)
        cout << *it << " ";
```

```
    l.remove(100);  // remove element
    cout << "\nAfter removing 100, List elements: ";
    for (int x : l)
        cout << x << " ";
    cout << endl;

    return 0;
}
```

Sample Output

```
Vector elements: 10 20 30
Size: 3, Capacity: 3
After pop_back, Vector elements: 10 20

List elements: 50 100 150
After removing 100, List elements: 50 150
```

EXPERIMENT 4

Aim:

To write a C++ program to implement a **Deque (Double-Ended Queue)** and perform various operations such as insertion and deletion from both ends.

Description:

A **Deque (Double Ended Queue)** is a linear data structure that allows insertion and deletion at both the **front** and the **rear** ends. Unlike a queue (FIFO) or stack (LIFO), deque provides more flexibility.
C++ STL provides the **deque container** that supports dynamic resizing and efficient insertion/deletion at both ends.

Common Operations on Deque:

- push_back() → Insert at rear
- push_front() → Insert at front
- pop_back() → Delete from rear
- pop_front() → Delete from front
- front() → Access first element
- back() → Access last element
- size() → Number of elements

Procedure:

1. Include the <deque> header file.
2. Declare a deque of type int.
3. Perform insertion using push_front() and push_back().
4. Perform deletion using pop_front() and pop_back().
5. Access elements using front() and back().
6. Display the deque contents.
7. Compile and run the program.

Program:

```
#include <iostream>
#include <deque>
```

```
using namespace std;

int main() {
    deque<int> dq;

    // Insert elements
    dq.push_back(10);   // Insert at rear
    dq.push_back(20);
    dq.push_front(5);   // Insert at front

    cout << “Deque elements: “;
    for (int x : dq) {
        cout << x << “ “;
    }
    cout << endl;

    // Access elements
    cout << “Front element: “ << dq.front() << endl;
    cout << “Back element: “ << dq.back() << endl;

    // Deletion
    dq.pop_front();  // Remove front element
    dq.pop_back();   // Remove rear element

    cout << “Deque after deletion: “;
    for (int x : dq) {
        cout << x << “ “;
    }
    cout << endl;
```

```
    // Insert more elements
    dq.push_front(100);
    dq.push_back(200);

    cout << "Deque final state: ";
    for (int x : dq) {
        cout << x << " ";
    }
    cout << endl;

    return 0;
}
```

Sample Output:

Deque elements: 5 10 20

Front element: 5

Back element: 20

Deque after deletion: 10

Deque final state: 100 10 200

EXPERIMENT 5

Aim

To write a C++ program that demonstrates the implementation of a **map** container and performs various operations such as insertion, traversal, search, deletion, and size check.

Description

- A **map** in C++ (from STL) is an **associative container** that stores elements in **key-value pairs**.
- Keys are **unique** and stored in **sorted order** (by default in ascending order).
- Values can be duplicated, but each key maps to a single value.
- Map operations include:
    - insert() → Add elements
    - erase() → Remove elements
    - find() → Search by key
    - size() → Number of elements
    - Iterators for traversal

Procedure

1. Start the program and include the required header <map>.
2. Declare a map with key and value types.
3. Insert key-value pairs using insert() or operator[].
4. Traverse the map using iterators and display its contents.
5. Perform search operation using find().
6. Delete an element using erase().
7. Display the size of the map.
8. End the program after successful execution.

Program

```
#include <iostream>
#include <map>
```

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

int main() {
    // Declare a map with key as int and value as string
    map<int, string> student;

    // Insertion
    student[1] = "Alice";
    student[2] = "Bob";
    student.insert({3, "Charlie"});
    student.insert({4, "David"});

    cout << "Map Elements (Key -> Value):" << endl;
    for (auto it = student.begin(); it != student.end(); ++it) {
        cout << it->first << " -> " << it->second << endl;
    }

    // Searching an element
    int key = 3;
    auto it = student.find(key);
    if (it != student.end()) {
        cout << "\nFound: " << it->first << " -> " << it->second << endl;
    } else {
        cout << "\nKey " << key << " not found." << endl;
    }

    // Deleting an element
    student.erase(2);
    cout << "\nAfter deleting key 2:" << endl;
```

```
    for (auto &pair : student) {
        cout << pair.first << " -> " << pair.second << endl;
    }

    // Size of the map
    cout << "\nSize of the map: " << student.size() << endl;

    return 0;
}
```

Sample Output

Map Elements (Key -> Value):

1 -> Alice

2 -> Bob

3 -> Charlie

4 -> David

Found: 3 -> Charlie

After deleting key 2:

1 -> Alice

3 -> Charlie

4 -> David

Size of the map: 3