WEEK 9

Program No:9.1

Develop a C++ program to demonstrate the use of virtual functions to achieve dynamic dispatch and enable runtime polymorphism.

Aim: Develop a C++ program to demonstrate the use of virtual functions to achieve dynamic dispatch and enable runtime polymorphism.

Description:

This C++ program demonstrates runtime polymorphism using virtual functions. It defines a base class with a virtual method and derived classes that override it. A base class pointer is used to call the overridden methods, showcasing dynamic dispatch—where the function call is resolved at runtime based on the actual object type.

Syntax:

```
class Base {
    public:
            virtual void functionName();
            };

   class Derived : public Base {
        public:
              void functionName() override;
            };

int main() {
       Base* ptr;
      Derived obj;
      ptr = &obj;
    ptr->functionName();
}
```

ADITYA UNIVERSITY

Program:

```
#include<iostream>
using namespace std;
class shape
{
            public:
                        virtual void draw()
                        {
                                    cout<<"Draw a generic shape"<<endl;
                        }
};
class circle:public shape
{
     public:

void draw() override
     {
               cout<<"drawing a circle"<<endl;
     }
};
class rectangle:public
   shape
{
   public:
     void draw() override
     {
     cout<<"drawing a rectangle"<<endl;
     }
};
class triangle:public
   shape
{
   public:
     void draw() override
     {
     cout<<"drawing a triangle"<<endl;
     }
};
int main()
{
cout<<"Roll no:24B11AI439"<<endl;
   shape* shapePtr;
   circle c;
   shapePtr=&c;
   shapePtr->draw();
   rectangle r;
   shapePtr=&r;
   shapePtr->draw();
   triangle t;
   shapePtr=&t;
   shapePtr->draw();
```

ADITYA
UNIVERSITY

```
        shape s;
        shapePtr=&s;
        shapePtr->draw();
        return 0;
    }
```

Output:

```
Roll no:24B11AI439
drawing a circle
drawing a rectangle
drawing a triangle
Draw a generic shape
```

ADITYA UNIVERSITY

Program No :9.2

Develop a C++ program that illustrates runtime polymorphism using virtual functions.

Aim : To develop a C++ program that illustrates runtime polymorphism using virtual functions

Description :

This C++ program illustrates runtime polymorphism using virtual functions. A base class declares a virtual method, and derived classes override it. A base class pointer is used to invoke the method, and due to dynamic dispatch, the correct derived class method is called at runtime, demonstrating polymorphic behavior.

Syntax :

```
class Base {
public:
   virtual void show();
};
class Derived : public Base {
public:
   void show() override;
};
int main() {
   Base* ptr;
   Derived obj;
   ptr = &obj;
   ptr->show();
}
```

ADITYA UNIVERSITY

Program:

```
#include <iostream>
using namespace std;
class Animal {
public:
   virtual void makeSound() {
      cout << "Animal makes a sound" << endl;
   }
};
```

```
class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Dog barks" << endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        cout << "Cat meows" << endl;
    }
};

int main() {

  cout<<"Roll no:24B11AI439"<<endl;
  Animal* animalPtr;
  Dog d;
  Cat c;
 animalPtr = &d;
 animalPtr->makeSound();
 animalPtr = &c;
 animalPtr->makeSound();
 return 0;
}
```

ADITYA UNIVERSITY

Output:

```
Roll no:24B11AI439
Dog barks
Cat meows
```

WEEK 10

Program No:10.1

Develop a C++ program that demonstrates the use of function templates to create functions that can work with different data types.

Aim: Develop a C++ program that demonstrates the use of function templates to create functions that can work with different data types.

Description:

Function templates in C++ allow you to write generic functions that work with any data type. This promotes code reusability and type safety by letting the compiler generate the appropriate function based on the type used during the call.

Syntax:

```
template <typename T>
void functionName(T arg1, T arg2) {
}
```

Program:

```
    #include<iostream>
    using namespace std;
    template<typename T>
    T mymax(T x, T y) {
        return (x > y) ? x : y;
    }
    template<typename T>
void swapValues(T &a,T&b) {
        T temp = a;
        a = b;
        b = temp;
    }
    int main() {
 cout<<"Rollno:24B11AI439"<<endl;
 cout << "Max of 3 and 7: " << mymax<int>(3, 7) << endl;
 cout << "Max of 3.0 and 7.0: " << mymax<double>(3.0, 7.0) << endl;
 cout << "Max of 'g' and 'e': " << mymax<char>('g', 'e') << endl;
 int a = 10, b = 20;
 cout << "\nBefore swap : a = " << a << ", b = " << b << endl;
        swapValues(a, b);
```

ADITYA UNIVERSITY

```
cout << "After swap : a = " << a << ", b = " << b << endl;
     double c = 15.0, d = 20.0;
 cout << "\nBefore swap : c = " << c << ", d = " << d << endl;
   swapValues(c, d);

   cout << "After swap : c = " << c << ", d = " << d << endl;
          char e = 'A', f = 'B';

    cout << "\nBefore swap : e = " << e << ", f = " << f << endl;
          swapValues(e, f);

    cout << "After swap : e = " << e << ", f = " << f << endl;
          return 0;

}
```

Output:

```
Roll no:24B11AI439
Max of 3 and 7: 7
Max of 3.0 and 7.0: 7
Max of 'g' and 'e': g
Before swap : a = 10, b = 20
After swap : a = 20, b = 10
Before swap : c = 15, d = 20
After swap : c = 20, d = 15
Before swap : e = A, f = B
After swap : e = B, f = A
```

ADITYA UNIVERSITY

Program No :10.2

Develop a C++ program that demonstrates template classes, which allow creating classes that can work with any data type

Aim : To develop a C++ program that demonstrates template classes, which allow creating classes that can work with any data type.

Description :

This C++ program uses template classes to create a generic Box class that can store and retrieve values of any data type—like int, double, or string. By using template , the class becomes flexible and reusable, eliminating the need to write separate classes for each type. It demonstrates type safety, code reuse, and the power of generic programming in C++.

Syntax :

```
template<class T>
class ClassName {
private:
      T data;
public:
    ClassName(T value)
{
data = value;
}
void display()
{
cout << "Data: " << data << endl;
}
};
// Creating objects for different data types
ClassName obj1(10);
ClassName obj2(5.5);
```

Program:

```
#include<iostream>
using namespace std;
template <class T>
class calculator
{
        private:
                T num1;
                T num2;
                public:
        calculator(T n1,T n2)
{
        num1=n1;
        num2=n2;
}
T add()
{
    return num1+num2;
}


T subtract()
{
    return num1-num2;
}
T multiply()
{
    return num1*num2;
}
T divide()
{
    if(num2!=0)
    return num1/num2;
    else
    {
        cout<<"Error!Division by zero."<<endl;
    }
}
};
int main()
{
    cout<<"Roll no:24B11AI439"<<endl;
    calculator<int> intcalc(10,5);
    cout<<"Int calculation:"<<endl;
    cout<<"Addition = "<<intcalc.add()<<endl;
    cout<<"subraction = "<<intcalc.subtract()<<endl;
    cout<<"Multiplication = "<<intcalc.multiply()<<endl;
    cout<<"Division = "<<intcalc.divide()<<endl;
    cout<<"-----------------------"<<endl;

    calculator<double> doublecalc(11.5,5.5);
    cout<<"double calculation:"<<endl;
    cout<<"Addition = "<<doublecalc.add()<<endl;
    cout<<"subraction = "<<doublecalc.subtract()<<endl;
    cout<<"Multiplication = "<<doublecalc.multiply()<<endl;
    cout<<"Division = "<<doublecalc.divide()<<endl;
    return 0;
}
```

ADITYA UNIVERSITY

Output:

```
Roll no:24B11AI439
Int calculation:
Addition = 15
subraction = 5
```

```
Multiplication = 50
```

```
Division = 2
------------------------
double calculation:
Addition = 17
subraction = 6
Multiplication = 63.25
Division = 2.09091
```

ADITYA UNIVERSITY

WEEK 11

Program No:11.1

Develop a C++ program that demonstrates exception handling using try, throw, and catch blocks.

Aim: To develop a C++ program that demonstrates exception handling using try, throw, and catch blocks.

Description:

Exception handling in C++ is a powerful feature that helps you deal with unexpected errors during program execution—like dividing by zero, accessing invalid memory, or failing to open a file.

- try block: Wraps the code that might cause an error.
- throw statement: Signals that an error has occurred and sends an exception.
- catch block: Receives and handles the exception, preventing the program from crashing.

This mechanism allows your program to respond gracefully to problems, display meaningful messages, and continue running or exit safely.

Syntax:

```
try {
   // Code that may cause an exception
   throw exception_value;  // Throwing an exception
}
catch (exception_type variable) {
   // Code to handle the exception
}
```

Program:

```
#include<iostream>
using namespace std;
int main()
{
cout<<"Roll no:24B11AI439"<<endl;
     int numerator,denominator;
   double result;
     cout<<"enter numerator:";
   cin>>numerator;
     cout<<"enter denominator:";
     cin>>denominator;
     try
{
```

ADITYA UNIVERSITY

```
    if(denominator==0)


    throw denominator;

        result=(double)numerator/denominator;
        cout<<"result ="<<result<<endl;
    }
    catch(int e)

    {

        cout<<"Error:Division by zero is not allowed!"<<endl;
    }
cout<<"end Program..."<<endl;
    return 0;


    }
```

Output 1:

```
Roll no:24B11AI439
enter numerator:5
enter denominator:0
Error:Division by zero is not allowed!
end Program...
```

Output 2:

```
Roll no:24B11AI439
enter numerator:2
enter denominator:5
result =0.4
end Program...
```

ADITYA
UNIVERSITY

Program No :10.2

Develop a C++ program to illustrate the use of multiple catch statements, where different types of exceptions are caught and handled differently.

Aim : Develop a C++ program to illustrate the use of multiple catch statements, where different types of exceptions are caught and handled differently.

Description :

This C++ program shows how to use multiple catch blocks to handle different types of exceptions. Each catch block is designed to respond to a specific error type—like int, char*, or std::exception—so the program can react appropriately based on what went wrong. This makes error handling more precise and flexible.

Syntax :

```
try {
   // Code that may throw different types of exceptions
   throw exception_value;
}
catch (int e) {
   // Handle integer exception
}
catch (const char* msg) {
   // Handle string literal exception
}
catch (const std::exception& ex) {
   // Handle standard exception
}
```

Program:

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
        cout<<"Roll no:24B11AI439"<<endl;
        int num1,num2;
        char op;
        cout<<"Simple calculator"<<endl;
```

ADITYA UNIVERSITY

```
        cout<<"enter first number:";

        cin>>num1;
        cout<<"enter second number:";
        cin>>num2;
        cout<<"Enter an operator(+,-,*,/):";
        cin>>op;
try
        {
                if(op!='+'&&op!='-'&&op!='*'&&op!='/')
                throw string("invalid operator!please use +,-,*,/.");
                if(num1<0||num2<0)
                throw -1;
                if(op!='/'&&num2==0)
                throw 0;
                double result;
                switch(op)
                {
                        case '+':result=num1+num2;
                        break;
                                case '-':result=num1-num2;
                        break;
                                case '*':result=num1*num2;
                        break;
                                case '/':result=num1/num2;
                        break;
  }



                cout<<"result : "<<result<<endl;
        }
        catch(int e)
        {
                cout<<"Error:Division by zero is not allowed!"<<endl;
        }
```

ADITYA UNIVERSITY

```
        catch(double e)
        {
```

```
cout<<"Error:Negative numbers are not allowed!"<<endl;
        }
        catch(string e)
        {
                cout<<"Error:"<<e<<endl;
        }
        cout<<"program execution completed successfully"<<endl;
        return 0;
}
```

Output:

```
Roll no:24B11AI439
Simple calculator
enter first number:3
enter second number:5
Enter an operator(+,-,*,/):+
result : 8
program execution completed successfully
```

ADITYA
UNIVERSITY

WEEK 12

Program No:12.1

Develop a C++ program of List and Vector Containers

Aim: To develop a C++ program List and Vector Containers.

Description:

In C++, a **vector** is a dynamic array that can automatically resize when elements are added or removed.

- ✓ **Random Access:** Access elements using [] or at().
- ✓ **Fast at End:** Insertion/deletion is fast at the end, slower in the middle.
- ✓ **Memory:** Stores elements in contiguous memory.
- ✓ **Functions:** push_back(), pop_back(), size().

A **list** is a doubly linked list where elements are stored in non-contiguous memory.

- ✓ **Fast Insertion/Deletion:** Anywhere in the list.
- ✓ **Sequential Access:** No random access, use iterators.
- ✓ **Memory:** Non-contiguous memory storage.
- ✓ **Functions:** push_back(), push_front(), pop_back(), pop_front().

ADITYA UNIVERSITY

Syntax:

```
 ===== VECTOR =====
vector<int> v;    // Declare vector
v.push_back(10);  // Add element at end
v.pop_back();     // Remove last element
v[0];             // Access element
v.size();         // Get size


// ===== LIST =====
list<int> l;      // Declare list
l.push_back(100); // Add element at end
l.push_front(200); // Add element at beginning
l.pop_back();     // Remove last element
l.pop_front();    // Remove first element
l.size();         // Get size
```

Program:

```
#include <iostream>
#include <list>
#include <vector>
using namespace std;
int main() {
    cout << "=== VECTOR OPERATIONS ===" << endl;
    vector<int> v; // declare a vector
    // Insertion
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    cout << "Vector elements after insertion: ";
    for (int x : v)
    cout << x << " ";
    // Deletion (remove last element)
    v.pop_back();
    cout << "\nVector after deletion: ";
    for (int x : v)
    cout << x << " ";
    // Access element
    cout << "\nFirst element: " << v.front();
    cout << "\nLast element: " << v.back() << endl;

    cout << "\n=== LIST OPERATIONS ===" << endl;
    list<int> lst; // declare a list
    // Insertion
    lst.push_back(100);
    lst.push_back(200);
    lst.push_front(50); // insert at beginning
    cout << "List elements after insertion: ";
    for (int x : lst)
    cout << x << " ";
    // Deletion
    lst.pop_front(); // remove first element
    cout << "\nList after deletion: ";
    for (int x : lst)
    cout << x << " ";
    // Traversal using iterator
    cout << "\nList traversal using iterator: ";
    for (list<int>::iterator it = lst.begin(); it != lst.end(); ++it)
    cout << *it << " ";
    cout << endl;
    return 0;
}
```

ADITYA UNIVERSITY

Output:

```
=== VECTOR OPERATIONS ===
Vector elements after insertion: 10 20 30
Vector after deletion: 10 20
First element: 10
Last element: 20
=== LIST OPERATIONS ===
 List elements after insertion: 50 100 200
List after deletion: 100 200
List traversal using iterator: 100 200
```

ADITYA UNIVERSITY

Program No:12.2

Develop a C++ program of Deque

Aim: To develop a C++ program of Deque.

Description:

□ A deque (double-ended queue) is a sequence container in C++ STL that allows insertion and deletion at both ends (front and back).
□ It is like a dynamic array, but more flexible than a vector for operations at the beginning.
□ Supports random access to elements using [] or at().
□ Useful when you need to add or remove elements from both ends efficiently.

- ✓ Can add/remove from front and back.
- ✓ Random access supported.
- ✓ Automatic resizing.
- ✓ Functions: push_back(), push_front(), pop_back(), pop_front(), size(), at(), [].

syntax:

```
{    deque<int> d;          // Declare a deque

    d.push_back(10);       // Add element at back
    d.push_front(20);      // Add element at front

    int x = d[0];          // Access element using index
    int y = d.at(1);       // Access element using at()

    d.pop_back();          // Remove element from back
    d.pop_front();         // Remove element from front

    int size = d.size();  // Get size
}
```

programm:

```
#include <iostream>
 #include <deque>
 using namespace std;
 int main() {
     deque<int> dq;
     cout << "=== DEQUE OPERATIONS ===" << endl;
     dq.push_back(10);
     dq.push_back(20);
     dq.push_front(5);
     cout << "Deque elements after insertion: ";
     for (int x : dq)
     cout << x << " ";
```

ADITYA UNIVERSITY

```
    dq.pop_front();
    cout << "\nDeque after deleting front element: ";
    for (int x : dq)
    cout << x << " ";
    dq.pop_back();
    cout << "\nDeque after deleting last element: ";
    for (int x : dq)
    cout << x << " ";
    cout << "\nFront element: " << dq.front();
    cout << "\nBack element: " << dq.back();
    dq.push_front(1);
    dq.push_back(50);
    cout << "\nDeque after adding 1 (front) and 50 (back): ";
    for (int x : dq)
    cout << x << " ";
    cout << endl;
    return 0;
}
```

Output:

```
=== DEQUE OPERATIONS ===
Deque elements after insertion: 5 10 20
Deque after deleting front element: 10 20
Deque after deleting last element: 10
Front element: 10 Back element: 10
Deque after adding 1 (front) and 50 (back): 1 10 50
```

ADITYA UNIVERSITY

program No:12.3

Develop a C++ program of Map and demonstrate operations such as insertion, deletion, access, and searching

Aim: To develop a C++ program of of Map and demonstrate operations such as insertion, deletion, access, and searching

Description:

A map is an associative container in C++ that stores key-value pairs with unique keys and keeps them sorted in ascending order. It allows fast access, retrieval, and modification of values using keys, making it ideal for situations where you need efficient lookups. Maps are usually implemented using balanced binary search trees, which ensures logarithmic time complexity for insertion, deletion, and searching.

Key Features and Operations:

- Insertion: Add elements using map[key] = value or insert().
- Access: Retrieve or update values using map[key].
- Searching: Check if a key exists using find(key).
- Deletion: Remove a key-value pair using erase(key).
- Traversal: Display all elements in sorted order of keys.
- Useful for storing unique keys, fast lookups, and dynamic data management.

program:

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
  map <int, strings> students;
  cout << "=== MAP OPERATIONS ===" << endl;
  students[101] = "Alice"; students[102] = "Bob";
  students[103] = "Charlie";
  students.insert({104, "David"});
  cout << "Students after insertion:" << endl;
  for (auto x : students)
  cout << "Roll No: " << x.first << " Name: " << x.second << endl;
  cout << "\nAccess element with key 102: " << students[102] << endl;
  int key = 103;
  auto it = students.find(key);
  if (it != students.end())
    cout << "Found student with Roll No " << key << ": " << it->second << endl;
  else
    cout << "Student with Roll No " << key << " not found!" << endl;
  students.erase(101);
  cout << "\nAfter deleting key 101:" << endl;
  for (auto x : students)
  cout << "Roll No: " << x.first << " Name: " << x.second << endl;
  cout << "\nTotal students: " << students.size() << endl;
  return 0;
}
```

ADITYA UNIVERSITY

Output:

```
 === MAP OPERATIONS ===
Students after insertion:
Roll No: 101 Name: Alice
Roll No: 102 Name: Bob
Roll No: 103 Name: Charlie
Roll No: 104 Name: David
Access element with key 102: Bob
Found student with Roll No 103: Charlie
After deleting key 101:
Roll No: 102 Name: Bob
Roll No: 103 Name: Charlie
Roll No: 104 Name: David Total students: 3
```

ADITYA UNIVERSITY