Objectives:

To understand the concept and role of virtual functions in C++ for achieving runtime polymorphism; to learn the need for virtual functions, use of base class pointers to access derived class objects, array of base class pointers, pure virtual functions and abstract classes, virtual destructors, reinterpret cast operator, and Run-Time Type Information (RTTI)

Theory:

# Virtual Functions

A virtual function is a member function in a base class that you expect to be overridden in derived classes. It ensures that the correct function is called for an object, regardless of the type of reference used for the function call.

Syntax:

|  |
| --- |
| class Base { public:  virtual void display(); // Virtual function }; |

Example:

|  |
| --- |
| #include <iostream>  using namespace std;    class Base { public:  virtual void show() { cout << "Base class" << endl; }  };    class Derived : public Base { public:  void show() override { cout << "Derived class" << endl;  }  };    int main() { Base\* ptr; Derived d; ptr = &d; ptr->show(); // Output: Derived class return 0;  } |

# Need of Virtual Function

Without virtual functions, function calls are resolved at compile time (static binding). Virtual functions enable runtime polymorphism (dynamic binding), allowing correct method calls based on the actual object type at runtime.

It is used when you want to override a function in a derived class and call it through a base class pointer or reference.

# Pointer to Derived Class

A base class pointer can point to a derived class object. If the base class function is virtual, the derived class version is invoked.

Example:

|  |
| --- |
| class Base { public:  virtual void greet() { cout << "Hello from Base" << endl;  }  };    class Derived : public Base { public:  void greet() override { cout << "Hello from Derived" << endl;  }  };    int main() { Base\* ptr; Derived d; ptr = &d; ptr->greet(); // Output: Hello from Derived return 0;  } |

# Array of Pointers to Base Class

You can create an array of base class pointers, each pointing to derived class objects.

This is useful in managing collections of objects polymorphically.

Example:

|  |
| --- |
| class Base { public:  virtual void print() {  cout << "Base" << endl;  }  };    class Derived1 : public Base { public:  void print() override {  cout << "Derived1" << endl;  }  };    class Derived2 : public Base { public:  void print() override {  cout << "Derived2" << endl;  }  };    int main() { Base\* arr[2];  Derived1 d1; Derived2 d2; arr[0] = &d1; arr[1] = &d2;    for(int i = 0; i < 2; i++) { arr[i]->print();  }  // Output:  // Derived1 // Derived2 return 0;  } |

# Pure Virtual Functions and Abstract Class

A pure virtual function is declared by assigning 0 in the base class. A class containing at least one pure virtual function is called an abstract class.

Syntax:

|  |
| --- |
| class Shape { public:  virtual void draw() = 0; // Pure virtual function  }; |

Example:

|  |
| --- |
| class Shape { public:  virtual void draw() = 0; // Pure virtual  };    class Circle : public Shape { public:  void draw() override { cout << "Drawing Circle" << endl;  }  };    int main() { Shape\* s; Circle c; s = &c; s->draw(); // Output: Drawing Circle return 0;  } |

# Virtual Destructors

A virtual destructor ensures that when a base class pointer deletes a derived class object, the derived class’s destructor is also called.

Syntax:

|  |
| --- |
| class Base { public:  virtual ~Base(); // Virtual destructor }; |

Example:

|  |
| --- |
| class Base { public:  virtual ~Base() { cout << "Base Destructor" << endl;  }  };    class Derived : public Base { public:  ~Derived() {  cout << "Derived Destructor" << endl;  } |
| };    int main() {  Base\* b = new Derived();  delete b;  // Output:  // Derived Destructor // Base Destructor  return 0;  } |

# reinterpret\_cast Operator

reinterpret\_cast is used to cast one pointer type to another, even if the types are unrelated. It is a low-level cast and should be used with caution.

Syntax:

Derived\* d = reinterpret\_cast<Derived\*>(b);

Example:

|  |
| --- |
| #include <iostream>  using namespace std;    int main() { int a = 65;  char\* ch = reinterpret\_cast<char\*>(&a); cout << \*ch << endl; // May print 'A' depending on system architecture return 0;  } |

# Run-Time Type Information (RTTI)

RTTI allows the type of an object to be determined during program execution using operators like typeid and dynamic\_cast.

Syntax:

typeid(object).name();

dynamic\_cast<Derived\*>(basePtr);

Example:

|  |
| --- |
| #include <iostream> #include <typeinfo>  using namespace std;    class Base { public:  virtual void func() {}  };    class Derived : public Base {};    int main() {  Base\* b = new Derived;  if (typeid(\*b) == typeid(Derived)) {  cout << "Object is of type Derived" << endl;  } delete b; return 0;  } |

Q1) Create a base class Shape with a virtual function draw() that prints "Drawing Shape". Derive two classes, Circle and Rectangle, each overriding draw() to print "Drawing Circle" and "Drawing Rectangle", respectively. In the main function:

* Create Circle and Rectangle objects.
* Use a Shape\* pointer to call draw() on both objects to show polymorphic behavior.
* Create a version of Shape without the virtual keyword for draw() and repeat the experiment.  Compare outputs to explain why virtual functions are needed.
* Use a Circle\* pointer to call draw() on a Circle object and compare with the base class pointer’s behavior.

Code:

|  |
| --- |
| #include <iostream> using namespace std; class Shape { public:  virtual void draw() { cout << "Drawing Shape" << endl;  }  };    class Circle : public Shape { public:  void draw() override { cout << "Drawing Circle" << endl;  }  };    class Rectangle : public Shape { public:  void draw() override { cout << "Drawing Rectangle" << endl;  }  };    int main() {  Circle circle;  Rectangle rectangle;  Shape\* shape1 = &circle; Shape\* shape2 = &rectangle; shape1->draw(); shape2->draw();  Circle\* circlePtr = &circle; circlePtr->draw(); return 0;  } |

Output:

Drawing Circle

Drawing Rectangle

Drawing Circle

Q2) Create an abstract base class Animal with a pure virtual function speak() and a virtual destructor. Derive two classes, Dog and Cat, each implementing speak() to print "Dog barks" and

"Cat meows", respectively. Include destructors in both derived classes that print "Dog destroyed" and "Cat destroyed".

In the main function:

* Attempt to instantiate an Animal object (this should fail).
* Create Dog and Cat objects using Animal\* pointers and call speak().
* Delete the objects through the Animal\* pointers and verify that derived class destructors are called.
* Modify the Animal destructor to be non-virtual, repeat the deletion, and observe the difference.

Code:

|  |
| --- |
| #include <iostream>  using namespace std;    class Animal { public:  virtual void speak() = 0; virtual ~Animal() { cout << "Animal destroyed" << endl;  }  };    class Dog : public Animal { public:  void speak() override {  cout << "Dog barks" << endl;  }  ~Dog() {  cout << "Dog destroyed" << endl;  }  };    class Cat : public Animal { public:  void speak() override { cout << "Cat meows" << endl;  }  ~Cat() { cout << "Cat destroyed" << endl;  }  };    int main() {  Animal\* dog = new Dog();  Animal\* cat = new Cat();    dog->speak();  cat->speak(); |

delete dog; delete cat;

return 0; }

Output:

Dog barks

Cat meows

Dog destroyed

Animal destroyed

Cat destroyed

Animal destroyed

Q3) Create a base class Employee with a virtual function getRole() that returns a string "Employee". Derive two classes, Manager and Engineer, overriding getRole() to return "Manager" and "Engineer", respectively. In the main function:

* Create an array of Employee\* pointers to store Manager and Engineer objects.
* Iterate through the array to call getRole() for each object.
* Use dynamic\_cast to check if each pointer points to a Manager, and if so, print a bonus message (e.g., "Manager gets bonus").
* Use typeid to print the actual type of each object.

Code:

|  |
| --- |
| #include <iostream>  #include <string> #include <typeinfo>  using namespace std;    class Employee { public:  virtual string getRole() {  return "Employee";  }  virtual ~Employee() {}  };    class Manager : public Employee { public:  string getRole() override { return "Manager";  }  };    class Engineer : public Employee { public:  string getRole() override { return "Engineer";  }  };    int main() {  Employee\* employees[4]; employees[0] = new Manager(); employees[1] = new Engineer(); employees[2] = new Manager();  employees[3] = new Engineer();    for (int i = 0; i < 4; ++i) { cout << "Role: " << employees[i]->getRole() << endl;    Manager\* m = dynamic\_cast<Manager\*>(employees[i]); if (m) { cout << "Manager gets bonus" << endl;  } |
| cout << "Actual type: " << typeid(\*employees[i]).name() << endl; cout << endl;  }    for (int i = 0; i < 4; ++i) { delete employees[i];  }    return 0;  } |

Output:

Role: Manager

Manager gets bonus

Actual type: 7Manager

Role: Engineer

Actual type: 8Engineer

Role: Manager

Manager gets bonus

Actual type: 7Manager

Role: Engineer

Actual type: 8Engineer

Q4) Create a class Student with an integer id and a string name. In the main function:

* Create a Student object.
* Use reinterpret\_cast to treat the Student object as a char\* and print its memory address.
* Use reinterpret\_cast to convert an integer (e.g., 100) to a pointer type and print it.

Code:

|  |
| --- |
| #include <iostream> #include <string>  using namespace std;    class Student { public: int id; string name;    Student(int i, string n) : id(i), name(n) {}  };    int main() {  Student s(1, "samdhi");    char\* rawMemory = reinterpret\_cast<char\*>(&s);  cout << "Memory address of Student object as char\*: " << static\_cast<void\*>(rawMemory) << endl;    int num = 100;  int\* ptr = reinterpret\_cast<int\*>(num); cout << "Integer 100 reinterpreted as pointer: " << ptr << endl;    return 0;  } |

Output:

Memory address of Student object as char\*: 0x61fecc

Integer 100 reinterpreted as pointer: 0x64

Q5) Create an abstract base class Vehicle with a pure virtual function operate() and a virtual destructor that prints "Vehicle destroyed". Derive two classes, Car and Truck, each implementing operate() to print distinct messages (e.g., "Car accelerates" and "Truck transports"). Include destructors in Car and Truck that print "Car destroyed" and "Truck destroyed", respectively.

In the main function:

* Create Car and Truck objects. Use Vehicle\* pointers to call operate() on both objects.
* Use a Car\* pointer to call operate() on a Car object and compare with the base class pointer’s behavior.
* Modify a copy of the Vehicle class to make operate() non-virtual, repeat the calls using base class pointers, and observe the output differences.
* Create an array of Vehicle\* pointers to store Car and Truck objects, then iterate to call operate() for each.
* Attempt to instantiate a Vehicle object to confirm it cannot be created.
* Delete the objects via Vehicle\* pointers to verify derived class destructor calls. Test again with a non-virtual destructor in a separate version and note the difference.
* Use reinterpret\_cast to treat a Car object as a char\* and print its memory address, then cast an integer (e.g., 1000) to a pointer type and print it.
* Apply dynamic\_cast to check if each pointer in the array points to a Car, printing "Car identified" if successful. Use typeid to display the actual type of each object.

Code:

|  |
| --- |
| #include <iostream>  #include <string>  #include <typeinfo>    using namespace std;    class Vehicle { public:  virtual void operate() = 0; virtual ~Vehicle() { cout << "Vehicle destroyed" << endl;  }  }; |

|  |
| --- |
| class Car : public Vehicle { public:  void operate() override { cout << "Car accelerates" << endl;  }  ~Car() {  cout << "Car destroyed" << endl;  } };  class Truck : public Vehicle { public:  void operate() override { cout << "Truck transports" << endl;  }  ~Truck() {  cout << "Truck destroyed" << endl;  } };  int main() {  Car\* c = new Car();  Truck\* t = new Truck();    Vehicle\* v1 = c;  Vehicle\* v2 = t;    v1->operate(); // Car accelerates v2->operate(); // Truck transports c->operate(); // Car accelerates    Vehicle\* vehicles[4]; vehicles[0] = new Car(); vehicles[1] = new Truck(); vehicles[2] = new Car();  vehicles[3] = new Truck();    for (int i = 0; i < 4; ++i) {  vehicles[i]->operate();    Car\* carPtr = dynamic\_cast<Car\*>(vehicles[i]); if (carPtr) { cout << "Car identified" << endl; }    cout << "Actual type: " << typeid(\*vehicles[i]).name() << endl;  }    for (int i = 0; i < 4; ++i) { delete vehicles[i];  }    delete c;  delete t;    Car myCar;  char\* memAddress = reinterpret\_cast<char\*>(&myCar); cout << "Car object memory address as char\*: " << static\_cast<void\*>(memAddress) << endl;    int number = 1000;  int\* fakePtr = reinterpret\_cast<int\*>(number);  cout << "Integer 1000 reinterpreted as pointer: " << fakePtr << endl; return 0;  } |

Output:

Car accelerates

Truck transports

Car accelerates

Car accelerates

Car identified

Actual type: 3Car

Truck transports

Actual type: 5Truck

Car accelerates

Car identified

Actual type: 3Car

Truck transports

Actual type: 5Truck

Car destroyed

Vehicle destroyed

Truck destroyed

Vehicle destroyed

Car destroyed

Vehicle destroyed

Truck destroyed

Vehicle destroyed

Car destroyed

Vehicle destroyed

Truck destroyed

Vehicle destroyed

Car object memory address as char\*: 0x61fed4

Integer 1000 reinterpreted as pointer: 0x3e8

Car destroyed

Vehicle destroyed