1. Introduction

In C++, a **virtual function** is a member function of a class that is declared using the keyword virtual and is meant to be **overridden in a derived class**.
Virtual functions support **runtime polymorphism**, which means the function call is resolved **at runtime** rather than compile time.

Virtual functions are an important concept of **Object-Oriented Programming (OOP)** and are mainly used in **inheritance**.

---

2. Polymorphism in C++

Polymorphism means **"many forms"**.
In C++, polymorphism is of two types:

1. **Compile-time polymorphism**
    - Function overloading
    - Operator overloading
2. **Runtime polymorphism**
    - Achieved using **virtual functions**

Virtual functions allow a **base class pointer** to call the **derived class version** of a function.

---

3. Need for Virtual Functions

Without virtual functions, C++ uses **early binding** (compile-time binding).
This causes the base class function to be called even when a derived class object is used.

Virtual functions solve this problem by using **late binding (dynamic binding)**.

Why virtual functions are required:

- To achieve **runtime polymorphism**
- To ensure **correct function execution**
- To allow **dynamic behavior**
- To support **object-oriented design**

---

4. Definition of Virtual Function

A **virtual function** is a member function of a class that:

- Is declared using the keyword virtual

- Is usually accessed using a **base class pointer**
- Is overridden in a derived class
- Is resolved at **runtime**

---

5. Syntax of Virtual Function

```
class Base {
public:
   virtual void display() {
      cout << "Base class display function";
   }
};
```

In the derived class:

```
class Derived : public Base {
public:
   void display() {
      cout << "Derived class display function";
   }
};
```

---

6. Example Without Virtual Function

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

class Base {
public:
   void show() {
      cout << "This is Base class show function";
   }
};

class Derived : public Base {
public:
   void show() {
      cout << "This is Derived class show function";
   }
};

int main() {
   Base* b;
   Derived d;
   b = &d;
   b->show();
   return 0;
}
```

Output

This is Base class show function

➡ This happens because function binding is done **at compile time**.

7. Example With Virtual Function

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

class Base {
public:
   virtual void show() {
      cout << "This is Base class show function";
   }
};

class Derived : public Base {
public:
   void show() {
      cout << "This is Derived class show function";
   }
};

int main() {
   Base* b;
   Derived d;
   b = &d;
   b->show();
   return 0;
}
```

Output

This is Derived class show function

➡ The correct function is called due to **runtime polymorphism**.

---

8. Working of Virtual Functions

Virtual functions work using a mechanism called **Virtual Table (V-Table)**.

V-Table (Virtual Table):

- Created by the compiler
- Contains addresses of virtual functions
- Each object of a class has a **pointer to the V-Table**
- Function calls are resolved using this table at runtime

---

9. Virtual Table (V-Table) Explanation

- Base class has a V-Table for its virtual functions
- Derived class creates its own V-Table if functions are overridden

- Base class pointer refers to derived object
- Compiler checks V-Table to find correct function

This ensures **dynamic binding**.

---

10. Rules for Virtual Functions

1. Must be a **member function** of a class
2. Cannot be **static**
3. Usually accessed using **base class pointer**
4. Declared using keyword virtual in base class
5. Derived class function must have **same signature**
6. Constructor cannot be virtual

---

11. Virtual Function and Inheritance

Virtual functions work only when **inheritance** is used.

- Base class pointer → derived class object
- Function call depends on **object type**, not pointer type

This is the key concept behind **runtime polymorphism**.

---

12. Pure Virtual Function

A **pure virtual function** is a virtual function that has **no definition** in the base class.

Syntax

```
virtual void display() = 0;
```

A class containing at least one pure virtual function is called an **abstract class**.

---

13. Example of Pure Virtual Function

```
class Shape {
public:
   virtual void draw() = 0;
};

class Circle : public Shape {
public:
   void draw() {
```

```
    cout << "Drawing Circle";
  }
};
```

➡ Abstract classes **cannot be instantiated**.

---

14. Virtual Destructor

When deleting an object using a base class pointer, the **destructor must be virtual** to avoid memory leaks.

```
class Base {
public:
  virtual ~Base() {
    cout << "Base Destructor";
  }
};
```

This ensures both **base and derived destructors** are called properly.

---

15. Advantages of Virtual Functions

- Achieves **runtime polymorphism**
- Ensures **correct function execution**
- Improves code flexibility
- Supports dynamic behavior
- Makes code more maintainable

---

16. Disadvantages of Virtual Functions

- Slightly slower due to runtime binding
- Requires extra memory for V-Table
- More complex than normal functions

---

17. Difference Between Virtual and Non-Virtual Functions

Feature	Virtual Function	Non-Virtual Function
Binding	Runtime	Compile-time
Polymorphism	Yes	No
Performance	Slower	Faster
Flexibility	High	Low

---

18. Common Mistakes

- Forgetting to declare base function as virtual
- Incorrect function signature in derived class
- Not using base class pointer
- Forgetting virtual destructor

---

19. Applications of Virtual Functions

- Game development
- GUI frameworks
- Simulation software
- Device drivers
- Polymorphic class hierarchies

---

20. Conclusion

Virtual functions are a **core feature of C++ OOP**. They enable **runtime polymorphism**, allowing a program to behave differently depending on the object type at runtime.

Using virtual functions:

- Improves design flexibility
- Supports dynamic behavior
- Makes large programs easier to manage

Mastering virtual functions is essential for becoming proficient in **C++ object-oriented programming**.