Objectives:

- To gain an understanding of Virtual Functions in C++.

Tools and Libraries Used:

- Programming Language: C++
- IDE: G++
- Libraries: include <iostream>, include <string>

Theory:

Virtual Functions:

Virtual functions are a core concept of Object-Oriented Programming (OOP) in C++ that enable runtime polymorphism. They allow a function defined in a base class to be overridden in derived classes. When a base class pointer or reference points to a derived class object, the correct overridden function (from the derived class) is executed at runtime.

Types of Virtual Functions:

A. Pure Virtual Functions:

These functions have no implementation in the base class and must be overridden in derived classes. A class containing at least one pure virtual function becomes an abstract class.

Syntax:

```
class Base {
public:
virtual void functionName() = 0;
};
```

B. Virtual Functions:

These are regular virtual functions that may have an implementation in the base class but can be overridden by derived classes.

Syntax:

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

Rules for Using Virtual Functions:

- Cannot be static: Since static functions belong to the class itself (not an instance), they do not support runtime polymorphism and cannot be virtual.
- Matching signatures: The function signature in the derived class must match the base class to ensure proper overriding and runtime dispatch.
- Access via pointers/references: Virtual functions are generally called through base class pointers or references to achieve runtime polymorphism.
- Friend functions: Virtual functions can be declared as friends of other classes. However, friend functions themselves cannot be virtual as they are not class members.
- Optional overriding: Derived classes are not required to override a virtual function. If they don't, the base class version will be executed.
- Constructors and destructors: Constructors cannot be virtual because objects are not fully established during construction. However, destructors should be virtual to ensure proper cleanup when deleting derived class objects via base class pointers.

Lab Questions:

Q no 1:

Create a base class Base with a virtual method display().
Create a derived class Derived that overrides the display() method.
Implement a main() function where you create a Base pointer pointing to a Derived object and call the display() method.

Code:

```
#include<iostream>
using namespace std;
class base{
    public:
    virtual void display () {
        cout<<"Displayed in base class";
    }
};
class derived : public base {
    public:
    void display() override {
        cout<<"Displayed in derived class";
    }
};
int main() {
    base *b;
    derived d;
    b=&d;
    b->display();
    return 0;
}
```

Output:

```
Displayed in derived class
```

Q no 2:

Create a base class Shape with a virtual destructor.
Create a derived class Circle that has a constructor and destructor.
Implement a main() function to demonstrate the use of virtual destructors by creating a Shape pointer pointing to a Circle object.

Code:

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

class shape {
public:
    virtual ~shape() {
        cout << "Destructor of class shape." << endl;
    }
};

class circle : public shape {
public:
    circle() {
        cout << "Constructor in class circle." << endl;
    }
    ~circle() {
        cout << "Destructor in class circle." << endl;
    }
};

int main() {
    shape* s = new circle();
    delete s;
    return 0;
}
```

Output:

```
Constructor in class circle.
Destructor in class circle.
Destructor of class shape.
```

Conclusion:

Understanding virtual functions is crucial for implementing polymorphism in C++. Through this lab, we learned how to:

- Define virtual functions to enable dynamic method binding.
- Override virtual functions in derived classes.
- Use pure virtual functions to create abstract classes.

These concepts form the foundation for designing flexible and reusable object-oriented systems. Mastery of virtual functions allows developers to write more efficient and adaptable code, making it easier to extend functionality in future developments.