1. Introduction

In C++, **function overloading** and **function overriding** are two important features of **Object-Oriented Programming (OOP)**.

- **Function overloading:** Same function name, **different parameters**, same scope.
- **Function overriding:** Same function name, **same parameters**, different class (base and derived).

These concepts improve **code readability, reusability, and maintainability**.

---

2. Need for Function Overloading & Overriding

- Avoids writing multiple function names for similar tasks
- Allows **flexible use of functions**
- Supports **polymorphism** in OOP
- Improves code organization
- Reduces errors and redundancy

---

3. Function Overloading

Function overloading is the process of **declaring multiple functions with the same name in the same scope but different parameter lists**.

Key Points:

- Functions must differ in **number of parameters or parameter types**
- Return type alone **cannot distinguish functions**
- Compile-time polymorphism (resolved at compile time)

---

4. Syntax of Function Overloading

```
return_type function_name(type1 arg1);
return_type function_name(type1 arg1, type2 arg2);
return_type function_name(type arg1, type arg2, type arg3);
```

---

5. Examples of Function Overloading

Example 1: Different number of parameters

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

int add(int a, int b) {
```

```
    return a + b;
}

int add(int a, int b, int c) {
    return a + b + c;
}

int main() {
    cout << add(5, 10) << endl;
    cout << add(5, 10, 15) << endl;
    return 0;
}
```

Output

```
15
30
```

Example 2: Different types of parameters

```
float add(float a, float b) {
    return a + b;
}

int add(int a, int b) {
    return a + b;
}
```

---

6. Rules for Function Overloading

1. Must have **same function name**
2. Must have **different parameter lists**
3. Return type **cannot be used alone** for overloading
4. Can be overloaded **inside the same class** or **in global scope**

---

7. Advantages of Function Overloading

- Improves code readability
- Allows **reuse of function names**
- Simplifies programming
- Supports **polymorphism at compile-time**

---

8. Function Overriding

Function overriding occurs when a **derived class provides its own definition of a base class function** with the **same name and parameters**.

Key Points:

- Requires **inheritance**
- Functions must have **same name and same parameter list**
- Supports **runtime polymorphism**
- Base class function must be **virtual** to achieve dynamic behavior

---

9. Syntax of Function Overriding

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

class Derived : public Base {
  public:
    void display() override { // override keyword optional
       cout << "Derived class display" << endl;
    }
};
```

---

10. Example of Function Overriding

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

class Base {
public:
   virtual void show() {
      cout << "Base class show" << endl;
   }
};

class Derived : public Base {
public:
   void show() {
      cout << "Derived class show" << endl;
   }
};

int main() {
   Base* b;
   Derived d;
   b = &d;
   b->show();  // Calls Derived's show() because of virtual
   return 0;
}
```

Output

Derived class show

---

11. Rules for Function Overriding

1. Function name **must be same**
2. Parameter list **must be same**
3. Return type **must be compatible** (C++11 supports covariant return type)
4. Access specifier can be **different** but usually public
5. Base class function should be **virtual** for runtime polymorphism

---

12. Overloading vs Overriding

Feature	Function Overloading	Function Overriding
Scope	Same class	Base and derived class
Parameters	Must differ	Must be same
Return Type	Can differ	Must be same or covariant
Compile/Runtime	Compile-time	Runtime (if virtual)
Inheritance required	No	Yes
Purpose	Convenience, readability	Polymorphism, runtime behavior

---

13. Function Overloading with Constructors

- Constructors can also be overloaded in C++
- Multiple constructors with different parameter lists provide flexible object initialization

```
class Student {
   int roll;
   float marks;
public:
   Student() { roll = 0; marks = 0; }
   Student(int r) { roll = r; marks = 0; }
   Student(int r, float m) { roll = r; marks = m; }
};
```

---

14. Virtual Functions and Overriding

- Virtual functions allow **runtime resolution** of overridden functions
- If base class function is not virtual, base class function is called by default

```
class Base {
   public:
      void show() { cout << "Base"; } // not virtual
};
```

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

Base* b = new Derived();
b->show(); // Output: Base
```

---

15. Advantages of Function Overriding

- Supports **runtime polymorphism**
- Enables **dynamic behavior** in inheritance
- Simplifies program design
- Allows base pointers to work with derived objects

---

16. Advantages of Function Overloading

- Reuse function names
- Improve code readability
- Reduce complexity
- Supports **compile-time polymorphism**

---

17. Common Mistakes

- Overloading functions by **only return type** (invalid)
- Forgetting **virtual keyword** for overriding
- Misusing base class pointers
- Overriding functions with **different parameters**

---

18. Best Practices

- Use overloading for **compile-time flexibility**
- Use overriding for **runtime polymorphism**
- Use `override` keyword in C++11+ for safety
- Keep function names meaningful
- Avoid excessive overloading or deep inheritance

---

19. Applications

- **Overloading:** Math operations, printing, constructors
- **Overriding:** GUI frameworks, polymorphic behavior, simulation systems
- Both concepts are essential in **OOP design** for **flexible and maintainable software**

---

20. Conclusion

Function overloading and function overriding are **powerful OOP features** in C++.

- **Overloading:** Same function name, different parameters, compile-time polymorphism
- **Overriding:** Same function name & parameters, base and derived class, runtime polymorphism

Mastering these concepts helps **write flexible, readable, and reusable code** in C++.