Month – 02 , class – 01

OOP c++ class – 06

Function overloading & Single Inheritance

1. Function Overloading

Function overloading means using the same function name with different parameter lists.
The compiler decides which one to call based on the arguments passed.
This is called compile-time polymorphism.

```
int fun() {
    cout<<"Hello World"<<endl;
    return 1;
}

char fun(int x, char c) {
    cout<<"The value of c and x are : "<<x<<" "<<c<<endl;
    return c;
}

void fun(int x) {
    x++;
    cout<<"The value of x = "<<x<<endl;
}
```

Here, all functions share the same name but have different parameters.
When we call them with different arguments, the correct version executes.

2. Ambiguity Error

If two functions have the same parameter list but different return types, the compiler becomes confused.

```
double fun(int a) { return 1.00; }
double fun(char a) { return 1.00; }
```

The `fun(int a)` version causes an ambiguity because both integer-based functions can match the same call.
The `fun(char a)` version works fine because the argument type is clearly different.

3. Conditions for Function Overloading

Function names must be the same.
Parameter list must differ in number or type.
Return type does not affect overloading.

4. Default Arguments with Overloading

When default arguments are used, overloading still works — but it must not create confusion.

```
void f(int x=0, int y=0) {
    cout<<"x = "<<x<<", y = "<<y<<endl;
}
```

Calling `f()`, `f(1)`, or `f(1,99)` will each call the same function, filling defaults where needed.

5. Ambiguity with Default Parameters

Sometimes default parameters make it unclear which function to call.

```
void a(int i, int x, char c='m') {
    cout<<i<<" "<<c<<endl;
}

void a(int x) {
    cout<<x<<endl;
}
```

Call `a(1,'s')` → works
Call `a(99)` → error (compiler can’t decide which one to choose)

6. Automatic Type Conversion Ambiguity

```
void car(float x) { cout<<x<<endl; }
void car(double x) { cout<<x<<endl; }

car(10); // ambiguous — int can convert to both float and double
```

When the compiler finds two possible matches after automatic conversion, it throws an ambiguity error.

7. Single Inheritance (Public)

One class derives from another to reuse its code.
Private members are not inherited, but protected and public ones are.

```
class Parent {
    int age;
protected:
    int net_worth;
public:
    string name;

    void sets(string n,int a,int nw) {
        name = n; age = a; net_worth = nw;
    }

    void display() {
        cout<<name<<endl<<age<<endl<<net_worth<<endl;
    }
};

class Child : public Parent {
public:
    void print() {
        cout<<net_worth<<endl; // ok (protected)
        cout<<name<<endl;      // ok (public)
        cout<<"It's the derived class function"<<endl;
    }
};
```

`Child` inherits `sets()` and `display()` from `Parent`.
Private member `age` is not accessible directly.

8. Private Inheritance

In private inheritance, public and protected members of the base become private in the derived class.

```
class Child : private Parent {
public:
    void print() {
        cout<<"It's the derived class function"<<endl;
    }
};
```

Now, even `name` becomes private in `Child`.
So, using `c.name = "Tia";` outside the class causes an error.

9. Protected Inheritance

In protected inheritance, public and protected members of the base class become protected in the derived class.

```
class Child : protected Parent {
public:
    void print() {
        cout<<"It's the derived class function"<<endl;
    }
};
```

Members are inherited but can’t be accessed through objects in main().

10. Multilevel Inheritance

When a derived class becomes the base for another class.

```
class Child1 : public Parent {
public:
    void print() {
        cout<<"It's the derived class function"<<endl;
    }
};

class Child2 : public Child1 {
public:
    void extra() {
        cout<<"Child2 function"<<endl;
    }
};
```

`Child2` inherits everything from both `Parent` and `Child1`.

This shows inheritance happening across multiple levels.

11. Constructor and Destructor Order

Base class constructor runs first, then derived.

When destroyed, the order reverses — derived first, then base.

```
class Base {
public:
    Base() { cout<<"Base constructor"<<endl; }
    ~Base() { cout<<"Base destructor"<<endl; }
};

class Derived : public Base {
public:
    Derived() { cout<<"Derived constructor"<<endl; }
    ~Derived() { cout<<"Derived destructor"<<endl; }
};
```

Output order:

Base constructor

Derived constructor

Derived destructor

Base destructor