1. Class and Object

- **Class**: Blueprint for objects.
- **Object**: Instance of a class.

Code:

```
#include <iostream> // Includes input/output stream library.
using namespace std; // Allows usage of standard library without prefixing with 'std::'.

class Student { // Defines a class named 'Student'.
public: // Access specifier allowing public access to members.
    string name; // Public member variable to store the student's name.
    int age; // Public member variable to store the student's age.

    void display() { // Public member function to display student's details.
        cout << "Name: " << name << ", Age: " << age << endl; // Outputs name and age.
    }
};

int main() {
    Student s1; // Creates an object 's1' of type 'Student'.
    s1.name = "Alice"; // Sets the 'name' of 's1' to "Alice".
    s1.age = 20; // Sets the 'age' of 's1' to 20.
    s1.display(); // Calls the 'display' method to print 's1' details.
    return 0; // Ends the program.
}
```

Output:

```
Name: Alice, Age: 20
```

2. Encapsulation

- **Definition**: Wrapping data and methods into a single unit (class).
- **Access Specifiers**: `private`, `public`, `protected`.

Code:

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

class BankAccount { // Defines a class named 'BankAccount'.
```

```
private: // Private access specifier; only accessible within this class.
    int balance; // Private member variable to store balance.

public: // Public access specifier for outside interaction.
    BankAccount() : balance(0) {} // Constructor initializing 'balance' to 0.

    void deposit(int amount) { // Public method to deposit money.
        balance += amount; // Adds 'amount' to 'balance'.
    }

    int getBalance() { // Public method to get the current balance.
        return balance; // Returns the private 'balance'.
    }
};

int main() {
    BankAccount account; // Creates a 'BankAccount' object.
    account.deposit(1000); // Calls 'deposit' method to add 1000.
    cout << "Balance: " << account.getBalance() << endl; // Outputs the balance.
    return 0;
}
```

Output:

```
Balance: 1000
```

---

3. Inheritance

- **Definition**: Deriving a class from another class.
- Types: `Single`, `Multilevel`, `Multiple`, etc.

Code:

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

class Animal { // Base class 'Animal'.
public: // Public access specifier.
    void eat() { // Method to indicate eating behavior.
        cout << "This animal eats food." << endl;
    }
};

class Dog : public Animal { // 'Dog' class inherits from 'Animal'.
public:
    void bark() { // Method to indicate barking behavior.
        cout << "The dog barks." << endl;
```

```
    }
};

int main() {
    Dog dog; // Creates a 'Dog' object.
    dog.eat(); // Calls the inherited 'eat' method from 'Animal'.
    dog.bark(); // Calls the 'bark' method of 'Dog'.
    return 0;
}
```

Output:

```
This animal eats food.
The dog barks.
```

---

4. Polymorphism

- **Definition**: Same name, different behavior.
- **Types**:
  - **Compile-time (Method Overloading)**.
  - **Runtime (Method Overriding)**.

Compile-time Polymorphism (Method Overloading)

Code:

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

class Calculator { // Defines a class named 'Calculator'.
public:
    int add(int a, int b) { // Overloaded method: Adds two integers.
        return a + b;
    }

    double add(double a, double b) { // Overloaded method: Adds two doubles.
        return a + b;
    }
};

int main() {
    Calculator calc; // Creates a 'Calculator' object.
    cout << "Int Addition: " << calc.add(5, 3) << endl; // Calls the integer version.
    cout << "Double Addition: " << calc.add(2.5, 3.5) << endl; // Calls the double version.
```

```
    return 0;
}
```

Output:

```
Int Addition: 8
Double Addition: 6
```

Runtime Polymorphism (Method Overriding)

Code:

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

class Animal { // Base class 'Animal'.
public:
    virtual void sound() { // Virtual method for sound behavior.
        cout << "This is a generic animal sound." << endl;
    }
};

class Dog : public Animal { // Derived class 'Dog'.
public:
    void sound() override { // Overrides the base class 'sound' method.
        cout << "The dog barks." << endl;
    }
};

int main() {
    Animal* animal; // Pointer of base class type.
    Dog dog; // Creates a 'Dog' object.
    animal = &dog; // Points 'animal' to 'dog'.
    animal->sound(); // Calls the overridden 'sound' method.
    return 0;
}
```

Output:

```
The dog barks.
```

---

5. Abstraction

- **Definition**: Showing essential details and hiding implementation.

Code:

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

class Shape { // Abstract base class.
public:
    virtual void area() = 0; // Pure virtual function for area calculation.
};

class Circle : public Shape { // Derived class 'Circle'.
    double radius; // Private member variable for radius.

public:
    Circle(double r) : radius(r) {} // Constructor initializes 'radius'.

    void area() override { // Implements the 'area' method.
        cout << "Circle Area: " << 3.14 * radius * radius << endl;
    }
};

int main() {
    Circle circle(5.0); // Creates a 'Circle' object with radius 5.0.
    circle.area(); // Calls the 'area' method to compute area.
    return 0;
}
```

Output:

```
Circle Area: 78.5
```

---

6. Constructor and Destructor

- **Constructor**: Initializes an object.
- **Destructor**: Cleans up resources.

Code:

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

class Student { // Defines a class named 'Student'.
public:
    Student() { // Constructor definition.
```

```
        cout << "Constructor Called!" << endl;
    }

    ~Student() { // Destructor definition.
        cout << "Destructor Called!" << endl;
    }
};

int main() {
    Student s1; // Creates a 'Student' object, invoking the constructor.
    return 0; // When 's1' goes out of scope, the destructor is called.
}
```

Output:

```
Constructor Called!
Destructor Called!
```

---

7. Operator Overloading

- **Definition**: Redefining operators for user-defined types.

Code:

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

class Complex { // Defines a class for complex numbers.
    int real, imag; // Private member variables.

public:
    Complex(int r, int i) : real(r), imag(i) {} // Constructor initializes variables.

    Complex operator+(const Complex& c) { // Overloads '+' operator.
        return Complex(real + c.real, imag + c.imag); // Adds real and imaginary parts.
    }

    void display() { // Displays the complex number.
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(1, 2); // Creates two complex numbers.
    Complex c3 = c1 + c2; // Uses overloaded '+' operator.
    c3.display(); // Displays the result.
    return 0;
```

```
}
```

Output:

```
4 + 6i
```

---

8. Friend Function

- **Definition**: A function that can access private/protected members of a class.

Code:

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

class Box { // Defines a class 'Box'.
private:
    int length; // Private member variable.

public:
    Box(int l) : length(l) {} // Constructor initializes 'length'.

    friend int getLength(Box b); // Friend function declaration.
};

int getLength(Box b) { // Friend function definition.
    return b.length; // Accesses private member 'length'.
}

int main() {
    Box box(10); // Creates a 'Box' object with length 10.
    cout << "Length: " << getLength(box) << endl; // Calls the friend function.
    return 0;
}
```

Output:

```
Length: 10
```

---

9. This Pointer

- **Definition**: Refers to the current object.

Code:

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

class Student { // Defines a class named 'Student'.
    int id; // Private member variable.

public:
    Student(int id) { // Constructor takes 'id' as input.
        this->id = id; // Uses 'this' pointer to distinguish member and parameter.
    }

    void display() { // Method to display 'id'.
        cout << "Student ID: " << id << endl;
    }
};

int main() {
    Student s(101); // Creates a 'Student' object with id 101.
    s.display(); // Calls the 'display' method.
    return 0;
}
```

Output:

```
Student ID: 101
```

---

10. Static Members

- **Static Variable**: Shared across all objects.
- **Static Function**: Operates on static data.

Code:

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

class Counter { // Defines a class named 'Counter'.
    static int count; // Static member variable shared by all objects.

public:
    Counter() { // Constructor increments 'count'.
        count++;
    }
```

```
    static int getCount() { // Static method to access 'count'.
        return count;
    }
};

int Counter::count = 0; // Initializes static member variable.

int main() {
    Counter c1, c2, c3; // Creates three 'Counter' objects.
    cout << "Count: " << Counter::getCount() << endl; // Calls static method.
    return 0;
}
```

Output:

```
Count: 3
```