1. Arrow Operator (->) in C++

The arrow operator -> is used to access **members (variables or functions) of a class or structure** through a **pointer** to an object.

Explanation:

- If `obj` is an object, normally you access members using dot `.`:
- `obj.member;`
- If you have a **pointer** to an object, `ptr`, you use ->:
- `ptr->member;  // equivalent to (*ptr).member`

Example:

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

class Student {
public:
    int roll;
    void display() {
        cout << "Roll number: " << roll << endl;
    }
};

int main() {
    Student s1;
    s1.roll = 101;

    Student* ptr = &s1;  // pointer to object
    ptr->display();      // using arrow operator

    return 0;
}
```

Output:

```
Roll number: 101
```

Key Point:

`ptr->member` is equivalent to `(*ptr).member`. Arrow operator is just a shorthand for dereferencing pointer and accessing member.

2. Array of Objects in C++

An **array of objects** is a collection of objects of the same class stored in contiguous memory locations. You can access each object using **array index**.

Explanation:

- Each element of the array is an **object**.

- You can access members of objects using **dot operator** or **arrow operator** (if pointer to array element).

Example 1: Using dot operator

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

class Student {
public:
    int roll;
    void display() {
        cout << "Roll number: " << roll << endl;
    }
};

int main() {
    Student students[3];  // array of 3 objects

    // Assign roll numbers
    students[0].roll = 101;
    students[1].roll = 102;
    students[2].roll = 103;

    // Display roll numbers
    for(int i = 0; i < 3; i++) {
        students[i].display();
    }

    return 0;
}
```

Output:

```
Roll number: 101
Roll number: 102
Roll number: 103
```

Example 2: Using pointer and arrow operator

```
Student* ptr = students;  // pointer to first object
ptr->roll = 201;           // first object
(ptr+1)->roll = 202;       // second object
(ptr+2)->roll = 203;       // third object
```

Key Points:

1. Arrays of objects allow you to store multiple objects together.
2. You can use loops to **initialize** or **display** object data.
3. You can use **pointer arithmetic** with arrays of objects.

`this` Pointer in C++

The `this` pointer is an **implicit pointer** available **inside all non-static member functions** of a class.
It **points to the object** that invoked the member function.

Key Points:

1. `this` is **automatically passed** as a hidden argument to all non-static member functions.
2. It **cannot be used in static member functions**, because static functions are **not tied to any object**.
3. Commonly used to:
   - **Distinguish between data members and parameters** when they have the same name.
   - **Return the current object** from a member function to allow **chaining**.

Example 1: Resolving name conflict

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

class Student {
private:
    int roll;

public:
    void setRoll(int roll) {
        this->roll = roll;  // this->roll refers to object member, roll
refers to parameter
    }

    void display() {
        cout << "Roll number: " << roll << endl;
    }
};

int main() {
    Student s1;
    s1.setRoll(101);
    s1.display();
    return 0;
}
```

Output:

```
Roll number: 101
```

Example 2: Returning current object (function chaining)

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

class Student {
private:
    int roll;

public:
    Student* setRoll(int r) {
        this->roll = r;
        return this;  // return current object
    }

    void display() {
```

```
        cout << "Roll number: " << roll << endl;
    }
};

int main() {
    Student s1;
    s1.setRoll(101)->display();  // chaining function calls
    return 0;
}
```

Output:

```
Roll number: 101
```

Polymorphism in C++

Polymorphism means **"many forms"**.
In C++, polymorphism allows **a single function, object, or operator** to behave differently in different contexts.

Polymorphism in C++ is mainly of **two types**:

Compile-Time (Static) Polymorphism

Occurs when the **function call is resolved at compile time**.

Types of Compile-Time Polymorphism:

- **Function Overloading**
- **Operator Overloading**

1. Function Overloading

- Multiple functions with **same name but different parameters**.
- Compiler decides which function to call based on **number or type of arguments**.

Example:

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

class Math {
public:
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
};

int main() {
    Math m;
    cout << m.add(5, 3) << endl;        // calls int version
```

```
    cout << m.add(5.5, 3.3) << endl;   // calls double version
    return 0;
}
```

2. Operator Overloading

- Redefining **operators** to work with **user-defined types**.

Example:

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

class Point {
public:
    int x, y;
    Point(int a, int b) { x = a; y = b; }
    Point operator+(Point p) {
        return Point(x + p.x, y + p.y);
    }
};

int main() {
    Point p1(2, 3), p2(3, 4);
    Point p3 = p1 + p2;  // uses overloaded +
    cout << "x = " << p3.x << ", y = " << p3.y << endl;
    return 0;
}
```

Run-Time (Dynamic) Polymorphism

Occurs when the **function call is resolved at runtime**.

Key Concept: Virtual Functions

- A function in **base class** is declared as **virtual** using `virtual` keyword.
- The function is **overridden** in the derived class.
- At runtime, **C++ decides which version to call** based on the object type.

Example:

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

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

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

```
int main() {
    Base* b;
    Derived d;
    b = &d;
    b->show();  // calls Derived's show() at runtime
    return 0;
}
```

Output:

```
Derived class function
```

Summary / Key Points:

1. **Polymorphism = Many forms of a single entity.**
2. **Compile-Time Polymorphism:** Function & Operator Overloading
   - Resolved at **compile time**
3. **Run-Time Polymorphism:** Virtual Functions
   - Resolved at **runtime** using **pointers or references**
4. Run-time polymorphism requires **inheritance** and **virtual functions**.

Virtual Function in C++

A **virtual function** is a **member function in a base class** that is declared with the keyword `virtual`.
It **allows derived classes to override it** and ensures that the **function call is resolved at runtime** (dynamic binding) rather than at compile time.

Key Points:

1. Declared using the keyword `virtual` in the **base class**.
2. **Enables Run-Time Polymorphism**.
3. The **type of object pointer/reference** determines which function is called at runtime.
4. Must be **member function**, **cannot be static**.
5. If a derived class does not override, **base class version is called**.

Example:

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

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

class Derived : public Base {
public:
    void show() {           // overrides base class function
        cout << "Derived class function" << endl;
    }
```

```
};

int main() {
    Base* b;      // base class pointer
    Derived d;

    b = &d;       // pointer points to derived object
    b->show();    // calls Derived's function at runtime

    return 0;
}
```

Output:

```
Derived class function
```

Without Virtual Function:

If `show()` is not virtual:

```
b->show();  // calls Base's function (compile-time binding)
```

Output:

```
Base class function
```

Abstract Class and Pure Virtual Function in C++

1. Abstract Class

An **abstract class** is a class that **cannot be instantiated directly** (you cannot create objects of it).
It is designed to be a **base class** for other derived classes.

It is created by including **at least one pure virtual function** inside the class.

Example:

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

class Shape {
public:
    virtual void area() = 0;   // Pure virtual function
};

class Circle : public Shape {
public:
    void area() {              // Function definition
        cout << "Area of Circle" << endl;
    }
};

int main() {
    // Shape s;       // Error: Cannot create object of abstract class
    Shape* s;         // Pointer to base class
```

```
    Circle c;
    s = &c;
    s->area();          // Calls Circle's version
    return 0;
}
```

Output:

```
Area of Circle
```

2. Pure Virtual Function

A **pure virtual function** is a **virtual function with no definition** in the base class.
It is declared by using `= 0` at the end of its declaration.

It acts as a **placeholder** that **must be overridden** by the derived class.

Example:

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

This means the base class **forces** derived classes to provide their own definition of `display()`.

3. Key Points:

- A class containing **at least one pure virtual function** becomes an **abstract class**.
- Abstract classes are used to **define common interfaces** for derived classes.
- You **cannot create objects** of an abstract class.
- You **can create pointers or references** of abstract class type.
- Derived classes **must override** pure virtual functions; otherwise, they also become abstract.

4. Example with Multiple Derived Classes

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

class Shape {
public:
    virtual void draw() = 0;  // pure virtual function
};

class Circle : public Shape {
public:
    void draw() {
        cout << "Drawing Circle" << endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() {
        cout << "Drawing Rectangle" << endl;
    }
```

```
};

int main() {
    Shape* s;
    Circle c;
    Rectangle r;

    s = &c;
    s->draw();

    s = &r;
    s->draw();

    return 0;
}
```

Output:

```
Drawing Circle
Drawing Rectangle
```