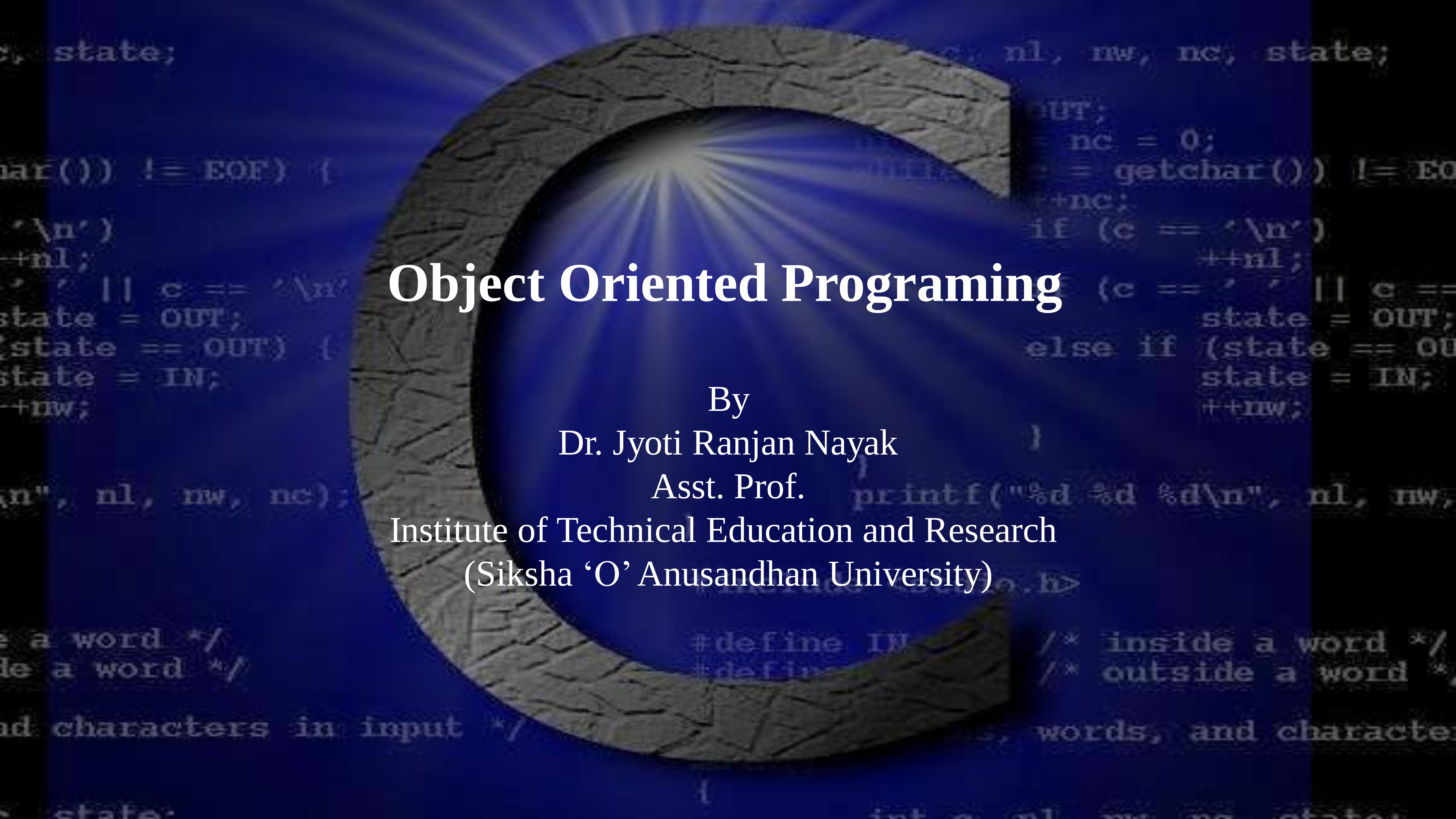


Object Oriented Programing

By
Dr. Jyoti Ranjan Nayak
Asst. Prof.
Institute of Technical Education and Research
(Siksha ‘O’ Anusandhan University)

Need of OOP:

- C++ language was designed with the main intention of adding object-oriented features to C language.
- As the size of the program increases, readability, maintainability and bug-free nature of programs decreases.
- This was the major problem with C language which relied upon functions of procedures.
- As a result, the possibility of not addressing the problem in an effective manner was high.
- Also, as data was almost neglected, data security was easily compromised.
- Using classes solves this problem by modeling program as a real world scenario.

Principles of Object Oriented Programming

- **Classes:-** Basic template for creating objects.
- **Objects:-** Basic run time entities.
- **Data abstraction & Encapsulation:-** Wrapping data and functions into single unit.
- **Inheritance:-** Properties of one class can be inherited into others.
- **Polymorphism:-** Ability to take more than one forms.
- **Dynamic Binding:-** Code which will execute is not known until the program runs.
- **Message Passing:-** Object.message (Information) call format.

CLASS

- ➢ A class is a template or a blueprint that binds the properties and functions of an entity or object.
- ➢ You can put all the entities or objects having similar attributes under a single roof, known as a class.
- ➢ When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Example:-

- Railway station having several trains.
- A train has some characteristics like train_no, destination, train_type, arrival_time, and departure_time. These are considered as data members of class.
- And its associated operations are arrival and departure. These operations are considered as the member functions of the class.

Syntax to Declare a Class:

```
class class_name
{
    // class definition
    access_specifier:
    data_members;
    member_functions()
          {
                //Body of function
           }
};
```

Access Specifiers:

- ➢ Access Specifiers in a class decide the accessibility of the class members, like variables or methods in other classes.
- ➢ It will decide whether the class members or methods will get directly accessed by the blocks present outside the class or not, depending on the type of Access Specifier.

There are three types of access modifiers in C++:

i. **Public:** Members are accessible from outside the class.

ii. **Private:** Members cannot be accessed (or viewed) from outside the class

iii. **Protected:** Members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

- ➢ Protected and Private data members or class methods can be accessed using a function only if that function is declared as the ***friend function***.

OBJECT

In C++, class variables are called as object.

Syntax:

class_name object_name;

```
#include <iostream>
#include<string>
using namespace std;
class student{
   public:
   char name[20];
   int id;
   float mark;
};
int main()
{
   student stud;     //Creat an object
   strcpy(stud.name, "JRN");
   stud.id=12;
   stud.mark=85.26;
   cout<<"Name: "<< stud.name<<endl;
   cout<<"Student ID: "<< stud.id<<endl;
   cout<<"Mark: "<< stud.mark<<endl;
   return 0;
}
```

Output:

```
Name: JRN
Student ID: 12
Mark: 85.26
```

By default access specifier in a class is private.

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
class student{
    char name[20];
    int id;
    float mark;
};
```

```
int main()
{
    student stud;     //Creat an object
    strcpy(stud.name, "JRN");
    stud.id=12;
    stud.mark=85.26;
    cout<<"Name: "<< stud.name<<endl;
    cout<<"Student ID: "<< stud.id<<endl;
    cout<<"Mark: "<< stud.mark<<endl;
    return 0;
}
```

error:

```
'char student::name [20]' is private within this context.

'int student::id' is private within this context.

'float student::mark' is private within this context
```

Member Function in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

Inside class definition

- **Syntax:**

```
class class_name{
public:
class_members;
return_type Method_name()  // method inside class definition
{
  // body of member function
}
};
```

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
class Person {
public:
    char name[20];
    int age;
void detail(){
        cout<<"Name is "<<name<<endl;
        cout<<"Age is: "<<age<<endl;
    }
};
```

```
int main()
{
    Person person1;
    strcpy(person1.name, "JRN");
    person1.age = 32;
    person1.detail();
    return 0;
}
```

Output:

```
Name is JRN
Age is: 32
```

Outside class definition

The member function is defined outside the class definition it can be defined using the scope resolution operator.

Syntax:

```
class class_name{
public:
class_members;
return_type method_name();
};
return_type class_name :: method_name()
{
  // body of member function
}
```

```
#include <iostream>
#include<string>
#include<cstring>
using namespace std;
class Person {
public:
   char name[20];
   int age;
   void detail();
};
void Person:: detail()
   {
      cout<<"Name is:
"<<name<<endl;
      cout<<"Age is: "<<age<<endl;
   }
```

```
int main()
{
   Person person1;
   strcpy(person1.name, "JRN");
   person1.age = 32;
   person1.detail();
   return 0;
}
```

Output:

```
Name is JRN
Age is: 32
```

Friend class

- A friend class can access the private and protected data of a class. We declare a friend class using the ***friend*** keyword inside the body of the class.

Syntax:

```
class class_name{
private/protected:
class_members;
friend class friend_class_name;
};
class friend_class_name;
{
  public:
  // body of member function
};
```

```
#include<iostream>
#include<cstring>
using namespace std;
//Friend Class
class detail;
class person
{
private:
    int age=32;
    char name[10]="JRN";
    friend detail;
};
```

```
class detail     //Friend Class
{
public:
    person p1;
    void display()
    {
        cout<<"Age : "<<p1.age<<endl;
        cout<<"Name : "<<p1.name;
    }
};
int main()
{
    detail obj;
    obj.display();
    return 0;
}
```

Output:

```
Age : 32
Name : JRN
```

Friend Function

- ➢ Like a friend class, a friend function can be granted special access to private and protected members of a class in C++.
- ➢ They are not the member functions of the class but can access and manipulate the private and protected members of that class for they are declared as friends.

A friend function can be:

i. A global function

ii. A member function of another class

i. Global function as friend function

Global Functions in C++ are the functions that are declared outside of any class, function or namespace. Global Functions are also called free functions.

Syntax:

friend return_type function_name (arguments);

```
#include<iostream>
#include<cstring>
using namespace std;
class person
{
   private:
   int age=32;
   char name[10]="JRN";
   public:
// declaration of Global function as a friend function
   friend void display(person);
};
```

```
// defining Global function
void display(person p)
{
   cout<<"Name "<<p.name<<endl;
   cout<<"Age "<<p.age<<endl;
}
int main()
{
   person p;
   display(p);
   return 0;
}
```

Output:

```
Name JRN
Age 32
```

ii Member function of another class

Member functions are the functions declared as a member of class. These functions do not include functions declared with friend specifiers.

Syntax:

friend return_type class_name::function_name (arguments);   // for a member function of another class

```
#include <iostream>
using namespace std;
class employee;
class organization {
        public:
        void memberFunction(employee& obj);
};
// base class for which friend is declared
class employee {
        private:
        int salary;
        protected:
        int emp_id;
        public:
        employee()
        {
                salary = 10;
                emp_id = 99;
        }
// friend function declaration
friend void organization::memberFunction(employee &);
};
// friend function definition
void organization::memberFunction(employee &obj)
{
cout << "Salary: " << obj.salary<< endl;
cout << "Employee ID: " << obj.emp_id;
}
int main()
{
employee object1;
organization object2;
object2.memberFunction(object1);
return 0;
}
```

Output:

```
Salary: 10
Employee ID: 99
```

Constructor:

A C++ class constructor is a special member function of a class responsible for initializing objects of that class.

Constructor can be overloaded.

Initialization– Constructors initialize objects of a class during their creation.

Automatic Invocation– They are automatically invoked when an object is created.

Same Name as Class– Constructors have the same name as the class and do not have a return type.

Return type- Constructors do not return value, hence they do not have a return type.

Syntax: (Declaration)

```
class ClassName {
public:
    ClassName(); // Constructor declaration
};
```

Syntax: (Definition inside the class)

```
class ClassName {
public:
    ClassName() { // Constructor definition within the class
        // Constructor body
    }
};
```

Syntax: (Definition outside the class)

```
ClassName::ClassName() {
    // Constructor definition outside the class
    // ClassName:: indicates the scope of the constructor
}
```

```
#include <iostream>
using namespace std;
class student {
public:
    string name;
    int roll;
    float cgpa;
    student() { // Constructor defined within the class
        name = "JRN";
        roll = 112;
        cgpa = 9.27;
    }
};
int main() {
    student s1;
    cout <<"Name: "<<s1.name<<endl;
    cout <<"Roll: "<<s1.roll<<endl;
    cout <<"CGPA: "<<s1.cgpa<<endl;
    return 0;
}
```

Output:

```
Name: JRN
Roll: 112
CGPA: 9.27
```

```
#include <iostream>
using namespace std;
class student {
public:
    string name;
    int roll;
    float cgpa;
    student();
};
// Constructor defined outside the class
student::student() {
    name = "JRN";
    roll = 112;
    cgpa = 9.27;
}
```

```
int main() {
    student s1;
    cout <<"Name: "<<s1.name<<endl;
    cout <<"Roll: "<<s1.roll<<endl;
    cout <<"CGPA: "<<s1.cgpa<<endl;
    return 0;
}
```

Constructor can be classified into 4 different categories.

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Move Constructor

1. Default Constructor

- ➢ A default constructor can be called with no arguments or one that doesn't have any parameters.
- ➢ It initializes the member variables of a class to their default values.
- ➢ **Syntax:**

```
class ClassName {
public:
    ClassName(); // Default constructor declaration
};
```

```
#include <iostream>
using namespace std;
class student {
public:
   string name;
   int roll;
   float cgpa;
// Default constructor
   student() {
      name = "JRN";
      roll = 112;
      cgpa = 9.27;
   }
};
int main() {
   student s1;
   cout <<"Name: "<<s1.name<<endl;
   cout <<"Roll: "<<s1.roll<<endl;
   cout <<"CGPA: "<<s1.cgpa<<endl;
   return 0;
}
```

Output:

```
Name: JRN
Roll: 112
CGPA: 9.27
```

2. Parameterized Constructor

- A parameterized constructor has parameters that allow the initialization of member variables with specific values passed during object creation.
- **Syntax:**

```
class ClassName {
public:
    ClassName(Type1 parameter1, Type2 parameter2, ...); // Parameterized constructor declaration
};
```

```
#include <iostream>
using namespace std;
class student {
public:
    string name="JRN";
    int roll=112;
private:
    float cgpa;
public:
    student(float cg) {
    cgpa = cg;
}
int display(){
    return cgpa;
}
};
```

```
int main() {
    student s1(9.27);
    cout <<"Name: "<<s1.name<<endl;
    cout <<“Roll: "<<s1.roll<<endl;
    cout <<"CGPA: "<<s1.display()<<endl;
    return 0;
}
```

Output:

```
Name: JRN
Roll: 112
CGPA: 9
```

3. Copy Constructor

- A copy constructor is a special member function that initializes a new object as a copy of an existing object of the same class.
- It is called when an object is passed by value, returned by value, or initialized with another object of the same class.
- **Syntax:**

```
class ClassName {
public:
    ClassName(const ClassName& obj); // Copy constructor declaration
};
```

```
#include <iostream>
using namespace std;
class Person {
public:
    string name;
    // Explicit copy constructor defined
    Person(const Person& obj) {
        name = obj.name;
    }
    Person(string n) {
        name = n;
    }
};
int main() {
    Person person1("JRN");
    Person person2 = person1; // Copying person1 to person2
    cout << "Name of person2: " << person2.name << endl;
    return 0;
}
```

Output:

```
Name of person2: JRN
```

4. Move Constructor

- Unlike the copy constructor, which duplicates an object in new memory, the move constructor leverages move semantics to transfer ownership of an existing object to a new one, bypassing redundant copies.

Syntax:

```
class ClassName {
public:
    ClassName(ClassName&& obj); // Move constructor declaration
};
```

```
#include <iostream>
class MoveExample {
private:
    int* data; // Pointer to dynamically allocated memory
public:
    // Constructor
    MoveExample(int value) {
        data = new int(value);
        std::cout << "Constructor: Allocating " << *data << std::endl;
    }
    // Copy Constructor
    MoveExample(const MoveExample& other) {
        data = new int(*other.data); // Deep copy
        std::cout << "Copy Constructor: Copying " << *data << std::endl;
    }
    // Move Constructor
    MoveExample(MoveExample&& other) noexcept {
        data = other.data;   // Transfer ownership
        other.data = nullptr; // Nullify the source to avoid dangling pointer
        std::cout << "Move Constructor: Moving data" << std::endl;
    }
};
```

```
int main() {
    MoveExample obj1(10); // Constructor
    MoveExample obj2 = std::move(obj1); // Move Constructor
    return 0;
}
```

Output:

```
Constructor: Allocating 10
Move Constructor: Moving data
```

Destructors

- Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.
- A destructor is a member function with the same name as its class prefixed by a ~ (tilde).
- **Syntax:**

```
class class_name {
public:
  // Constructor for class
  class_name();
  // Destructor for class
  ~ class_name();
};
```

```
#include <iostream>
using namespace std;
class Student
 {
   public:
        Student()
        {
            cout<<"Constructor Invoked"<<endl;
        }
        ~Student()
        {
            cout<<"Destructor Invoked"<<endl;
        }
};
int main(void)
{
    Student e1; //creating an object of Student
    Student e2; //creating an object of Student
    return 0;
}
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

Constructor Overloading

- The use of multiple constructors in the same class is known as Constructor Overloading.
- The constructor must follow one or both of the two rules below.
- All the constructors in the class should have a different number of parameters.
- It is also allowed in a class to have constructors with the same number of parameters and different data types.

```
#include <iostream>
#include<cstring>
using namespace std;
class Student {
    private:
        // Member Variable Declaration.
        string Name;
        int Age;
    public:
        // Constructor with no arguments.
        Student() {
            Name = "Rohan";
            Age = 23;
        }
        // Constructor with two arguments.
        Student(string str, int x) {
            Name = str;
            Age = x;
        }
// Member functions declaration.
        string get_Name() {
            return Name;
        }
        int get_Age() {
            return Age;
        }
};
int main() {
    Student stu1, stu2("Mohit", 25);
    cout<<"Name: "<<stu1.get_Name()<<" Age: "<<stu1.get_Age()<<endl;
    cout<<"Name: "<<stu2.get_Name()<<" Age: "<<stu2.get_Age()<<endl;
    return 0;
}
```

Output:

```
Name: Rohan Age: 23
Name: Mohit Age: 25
```

ABSTRACTION

- Abstraction in C++ is a fundamental Object-Oriented Programming (OOP) concept that focuses on hiding the implementation details and showing only the necessary features of an object. It allows users to interact with an object through a defined **interface** without knowing how it works internally.
- **Example:**

  When we drive a car and there is another vehicle coming from the front so we apply the brake. Now we are only concerned about applying the brake. We don’t go deep into how the brake mechanism works internally like the brake box, brake shoe, etc. This is what abstraction is

Types of abstraction

There are two types of abstraction:

1. **Control abstraction**
2. **Data abstraction**

1. Control abstraction:

- ➢ In this type of abstraction implementation, the process is hidden from the user.
- ➢ Control Abstraction is a technique for creating new features by combining control statements into a single unit.

Control abstraction in C++ is achieved through:

i. Function Abstraction (Procedural Abstraction)

ii. Iteration Abstraction (Loops & Iterators)

iii. Selection Abstraction (Conditional Statements)

iv. Concurrency Abstraction (Threads & Async Execution)

2. Data abstraction

- ➢ Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
- ➢ Data Abstraction can be achieved in two ways:

i. Abstraction using classes

ii. Abstraction in header files

i. Abstraction using classes

Data abstraction is implemented in C++ using access specifiers in classes:

private → Members are hidden from outside the class (used for abstraction).

public → Members are accessible from outside (used for the interface).

```
#include <iostream>
using namespace std;
class TestAbstraction {
   private: string x, y;
   public:
      void set(string a, string b) {
         x=a;
         y=b;
      }
      void print() {
      cout<<"x = "<<x<<endl;
      cout<<"y = "<<y<<endl;
   }
};
int main() {
   TestAbstraction t;
   t.set("Training", "class");
   t.print();
   return 0;
}
```

- In this example, abstraction is achieved using classes and access specifiers.
- TestAbstraction class is declared with string x, y as its private members, which means we could not access these strings outside the class.
- void set(string a, string b) and void print() are declared public members' functions.
- TestAbstraction t creates the object of TestAbstraction class.
- t1.set(“Training", “class"); sets the private member's string x as Training, y as class. These implementation details are hidden from the user.

Output:

```
x = Training
y = class
```

ii Abstraction in header files

- We achieve data abstraction using classes in header files (.h files) and defining their implementation in separate .cpp files.

calculator.h

```
#ifndef CALCULATOR_H
#define CALCULATOR_H
// Function declarations (interface)
int add(int a, int b);
int subtract(int a, int b);
#endif // CALCULATOR_H
```

calculator.cpp

```
#include "calculator.h"
// Function definitions (implementation)
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
```

main.cpp

```
#include <iostream>
#include "calculator.h"  // Include the header file
int main() {
    int x = 10, y = 5;
    std::cout << "Sum: " << add(x, y) << std::endl;
    std::cout << "Difference: " << subtract(x, y) << std::endl;
    return 0;
}
```

Output:

```
Sum: 15
Difference: 5
```

Pure Virtual Functions and Abstract Classes

- Pure virtual functions are used if a function doesn't have any use in the base class but the function must be implemented by all its derived classes.
- A pure virtual function doesn't have the function body and it must end with = 0.
- A class becomes abstract when it has at least one pure virtual function.
- An abstract class can have constructors.
- **Syntax:**

```
class abstract_class {
public:
virtual void pure_virtual_function() = 0;  // Pure virtual function
};
```

```
#include <iostream>
using namespace std;
// Abstract class
class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
};
// Derived class
class Circle : public Shape {
public:
    void draw() override {  // Implementing the pure virtual function
        cout << "Drawing a Circle" << endl;
    }
};
int main() {
    Shape* shape = new Circle();  // Pointer to abstract class
    shape->draw();  // Calls Circle's draw() method
    delete shape;  // Free allocated memory
    return 0;
}
```

Output:

```
Drawing a Circle
```

ENCAPSULATION

- ➢ Encapsulation in C++ is a key OOPs concept that entails bundling data and function members into a single unit, i.e., a class. Correct use of encapsulation can make the code more maintainable, secure, and reusable.
- ➢ The class visibility can be manipulated and modified using access specifiers (public, private, and protected).
- ➢ For example, if the class data is marked as private, anyone outside the class cannot access it.
- ➢ Alternatively, if the class functionality is designated as public, it allows access from outside the class.
- ▪ Encapsulation hides sensitive data and exposes only necessary operations.
- ▪ Getter methods retrieve private data without exposing it directly.
- ▪ Setter methods validate data before modifying it, ensuring integrity.

```
#include <iostream>
using namespace std;
class Student {
private:
    string name;
    int age;
public:
    // Setter method to assign values
    void setStudent(string studentName, int studentAge) {
        name = studentName;
        age = studentAge;
    }
    // Getter method to display values
    void displayStudent() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};
```

```
int main() {
    Student s1;
    // Setting values using setter method
    s1.setStudent("JRN", 35);
    // Displaying values using getter method
    s1.displayStudent();
    return 0;
}
```

Output:

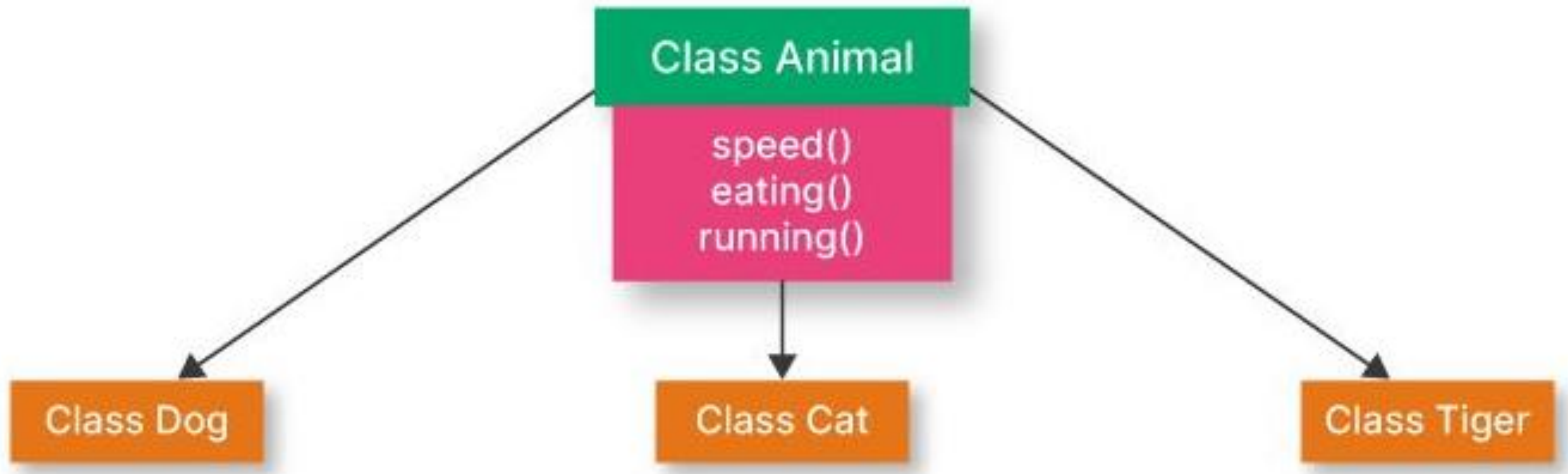
```
Name: JRN
Age: 35
```

INHERITANCE

- Inheritance in C++ is a process by which a new class can inherit the attributes and methods of an existing class.
- The existing class is then called the parent class, and the inheriting class is called the child class.
- Inheritance introduces the concepts of the child (derived) and parent (base) classes. That is:
- child class (derived) - the class that inherits from another class
- parent class (base) - the class being inherited from
- **Syntax:**

```
class Parent_Class_Name {
// parent class definition
};
class Child_Class_Name : access_specifier Parent_Class_Name {
// child class definition
};
```

- Private members of the base class never get inherited by the child class.
- If the public members of the base class are privately inherited by the child class, then they become the private members of the child class. Thus, the public members of the base class are only accessible by the member functions of the child class, not by the object of the child class.



```
#include <iostream>
using namespace std;
// Base class
class Animal {
public:
    void eat() {
        cout << "This animal eats food." << endl;
    }
};
// Derived class
class Dog : public Animal {
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};
```

```
int main() {
    Dog myDog;
    // Calling base class method
    myDog.eat();
    // Calling derived class
method
    myDog.bark();
    return 0;
}
```

Output:

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

Types of Inheritance in C++

C++ supports multiple types of inheritance:

1. Single Inheritance – A class inherits from a single base class.
2. Multiple Inheritance – A class inherits from more than one base class.
3. Multilevel Inheritance – A derived class inherits from another derived class.
4. Hierarchical Inheritance – Multiple derived classes inherit from a single base class.
5. Hybrid Inheritance – A combination of multiple types of inheritance.

1. Single Inheritance

In single inheritance, a derived class inherits from a single base class, i.e., a child class inherits from a single parent class.

Previous example is the example of single inheritance.

2. Multiple Inheritance

In multiple inheritance, a subclass inherits from multiple base classes, i.e., a child class inherits from more than one parent class.

Syntax

```
class B1 {
// Base class members
};
class B2 {
// Base class members
};
class DerivedClass : access_modifier B1, access_modifier B2 {
// Derived class members
};
```

```
#include <iostream>
using namespace std;
// First base class
class Engine {
    public:
    void start() {
        cout << "Engine started." << endl;
    }
};
// Second base class
class Wheels {
    public:
    void roll() {
        cout << "Wheels are rolling." << endl;
    }
};
```

```
// Derived class
class Car : public Engine, public
Wheels {
    public:
    void drive() {
        cout << "Car is moving." << endl;
    }
};
int main() {
    Car myCar;
    myCar.start(); // From Engine class
    myCar.roll();  // From Wheels class
    myCar.drive(); // From Car class
    return 0;
}
```

Output:

```
Engine started.
Wheels are rolling.
Car is moving.
```

3. Multilevel Inheritance

In multilevel inheritance, a derived class becomes the base class for another derived class.

Syntax:

```
class B1 {
// Base class 1 members
};
class B2 : access_modifer B1 {
// Base class 2 members
};
class DerivedClass : access_modifier B2 {
// Derived class members
};
```

```cpp
#include <iostream>
using namespace std;
// Base class
class Animal {
public:
    void eat() {
        cout << "This animal eats food." << endl;
    }
};
// Intermediate derived class
class Mammal : public Animal {
public:
    void breathe() {
        cout << "This mammal breathes air." << endl;
    }
};
// Final derived class
class Dog : public Mammal {
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};
int main() {
    Dog myDog;
    myDog.eat();     // From Animal class
    myDog.breathe(); // From Mammal class
    myDog.bark();    // From Dog class
    return 0;
}
```

Output:

```
This animal eats food.
This mammal breathes air.
The dog barks.
```

4. Hybrid inheritance

Hybrid inheritance in C++ is a combination of more than one inheritance type within the same program.

```
#include <iostream>
using namespace std;
// Base class
class A {
public:
    void showA() {
        cout << "Class A" << endl;
    }
};
// Derived class from A
class B : public A {
public:
    void showB() {
        cout << "Class B" << endl;
    }
};
// Another derived class from A
class C : public A {
public:
    void showC() {
        cout << "Class C" << endl;
    }
};
// Derived class from B and C (Hybrid Inheritance)
class D : public B, public C {
public:
    void showD() {
        cout << "Class D" << endl;
    }
};
int main() {
    D obj;
    obj.showB();
    obj.showC();
    obj.showD();
    return 0;
}
```

Output:

```
Class B
Class C
Class D
```

5. Hierarchical Inheritance

In hierarchical inheritance, multiple derived classes inherit from a single base class, i.e., more than one child class inherits from a single parent class.

Syntax:

```
class B {
// body of the class
}
class D1 : access_modifier B {
// body of the class
}
class D2 : access_modifier B {
// body of the class
}
class D3 : access_modifier B {
// body of the class
}
```

```
#include <iostream>
using namespace std;
// Base class
class Animal {
public:
    void eat() {
        cout << "This animal eats food." <<
endl;
    }
};
// Derived class 1
class Dog : public Animal {
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};
// Derived class 2
class Cat : public Animal {
public:
    void meow() {
        cout << "The cat meows." << endl;
    }
};
int main() {
    Dog dog;
    Cat cat;
    // Calling base class method from derived classes
    dog.eat();
    dog.bark();
    cat.eat();
    cat.meow();
    return 0;
}
```

Output:

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

POLYMORPHISM

Polymorphism means many forms. It is an object-oriented programming concept that refers to the ability of a variable, function, or object to take on multiple forms, which are when the behavior of the same object or function is different in different contexts.

```
#include<iostream>
#include<cstring>
using namespace std;
class poly{
    public:
    void func(int a, int b){
        cout<<"Addition in int= "<<a+b;
    }
    void func(double a, double b){
        cout<<"Addition in double= "<<a+b;
    }
    void func(char a, char b){
        cout<<"First character is "<<a;
        cout<<“\nSecond character is "<<b;
    }
};
```

```
int main()
{
    poly p;
    p.func(12,25);
    return 0;
}
```

Output: Addition in int= 37

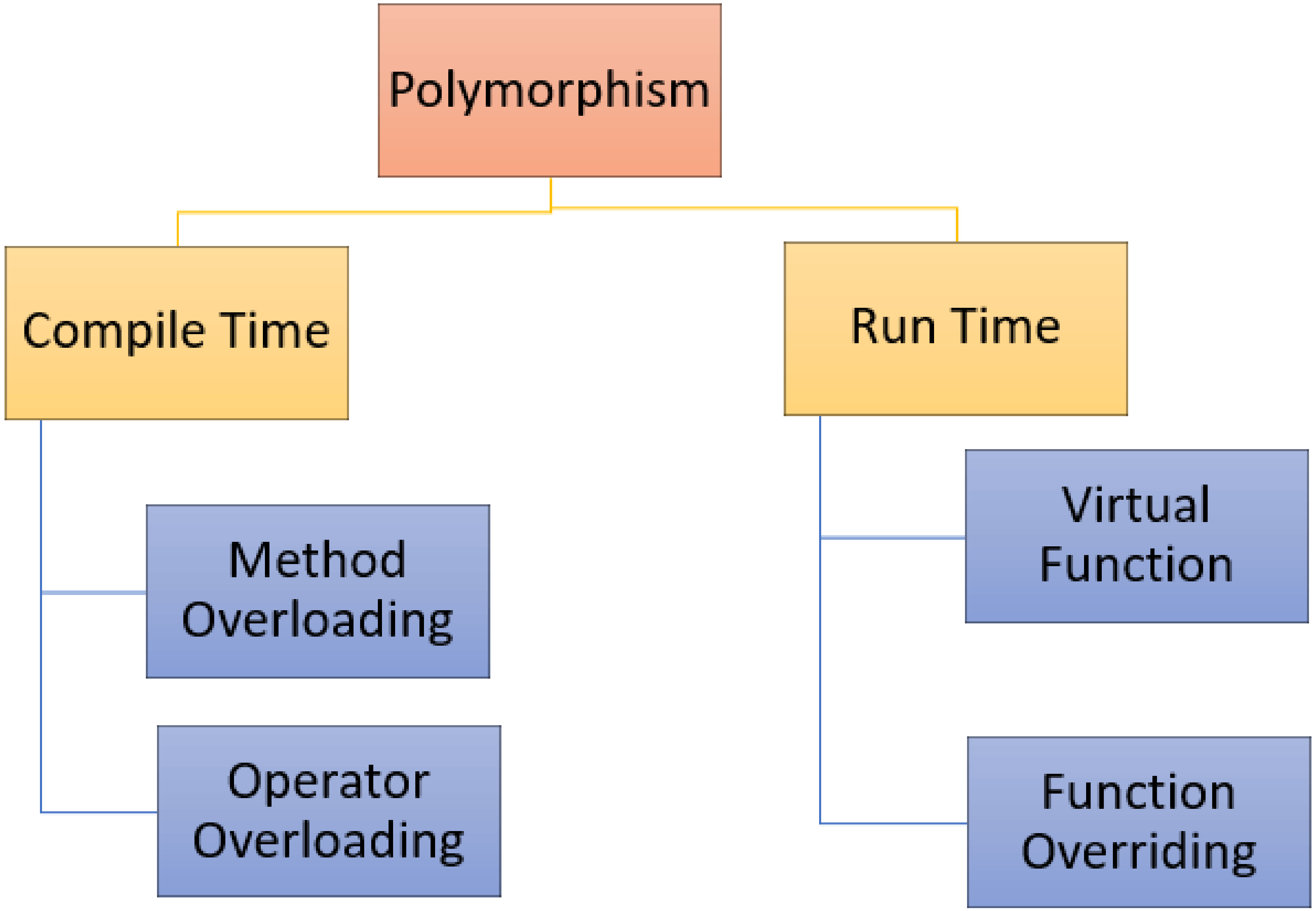
If we change the bold line as p.func(10.2,23.1) then

Output: Addition in double= 33.3

If we change the bold line as p.func(‘a’,’z’) then

Output:

```
First character is a
Second character is z
```


Polymorphism
Compile Time
Run Time
Method Overloading
Operator Overloading
Virtual Function
Function Overriding

Function/Method Overloading

Function overloading occurs when we have many functions with similar names but different arguments. The arguments may differ in terms of number or type.

```
#include<iostream>
#include<cstring>
using namespace std;
class poly{
    public:
    void add(int a, int b){
        cout<<"Addition = "<<a+b;
    }
    void add(int a, int b, int c){
        cout<<"Addition = "<<a+b+c;
    }
    void add(int a, int b, int c, int d){
        cout<<"Addition = "<<a+b+c+d;
    }
};
```

```
int main()
{
    poly p;
    p.add(10,20);
    return 0;
}
```

Output: *Addition = 30*

If we change the bold line as p.add(10,20,30) then

Output: *Addition in double= 60*

If we change the bold line as p.add(10,20,30,40) then

Output: *Addition in double= 100*

Operator Overloading

In C++ programming language, we can overload operators, i.e., make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is operator overloading.

Example of built-in data type :

```
int a;
float b, sum;
sum = a+b;
```

In the above example, the variables ‘a’ and ‘b’ are of types “int” and “float”, which are built-in data types. Hence the addition operator ‘+’ can easily add the contents of “a” and “b”. This is because the addition operator ‘+’ is predefined to add variables of built-in data type only.

Example of user-defined type:

```
class A
{
 //Statements;
};
int main()
{
      A   a1,a2,a3;
      a3= a1 + a2;
      return 0;
}
```

- Here we are trying to add two objects “a1” and “a2”, which are user-defined types i.e. of type “class A” using the “+” operator.
- This is not allowed, because the addition operator “+” is predefined to operate only on built-in data types. But here, “class A” is a user-defined type, so the compiler generates an error.
- This is where the concept of “Operator overloading” comes in.

```
#include<iostream>
using namespace std;
class A{
    public:
    int num;
    A(int n){
        num=n;
    }
};
int main()
{
    A a1(10),a2(a1++),a3(a1+a2);
    return 0;
}
```

Errors

Operators that can be overloaded

We can overload the following operators:

a. Unary operators

b. Binary operators

a. Unary operators:

A unary operator is an operator that works on a single operand. Overloading with these operators known as unary operator overloading.

```
#include <iostream>
using namespace std;
class increment {
   private:
    int num;
   public:
    // Constructor to initialize
    increment():num(12) {}
    // Overload ++
    increment operator ++() {
        num=num+1;
    }
    void display() {
        cout<<"Number is: "<<num<<endl;
    }
};
```

```
int main() {
    increment a;
    // Call the "operator ++()" function
    ++a;
    a.display();
    return 0;
}
```

Output:

```
Number is: 13
```

b. Binary operators:

A binary operator is an operator that works on two or more operands. Overloading with these operators known as binary operator overloading.

```
#include <iostream>
using namespace std;
class bin_operator {
   private:
    int num;
   public:
    // Constructor
    bin_operator(int n) {
        num=n;
    }
    // Overload +
    bin_operator operator+(bin_operator b) {
        bin_operator c(0);
        c.num=num+b.num;
        return c;
    }
    void display() {
        cout<<"Number is: "<<num<<endl;
    }
};
int main() {
    bin_operator a(10),b(20),sum(0);
    sum=a+b;
    sum.display();
    return 0;
}
```

Output:

```
Number is: 30
```

Note: We can not overload these following operators:

- :: scope resolution operator
- ?: conditional operator
- . Dot operator
- sizeof() operator

Virtual Function

- A virtual function is another way of implementing run-time polymorphism in C++. It is a special function defined in a base class and redefined in the derived class.
- To declare a virtual function, you should use the virtual keyword. The keyword should precede the declaration of the function in the base class.
- The virtual function in the base class helps the derived class method to override that function.

```
#include <iostream>
using namespace std;
class parent {
    public:
    void display() {
        cout<<"I am the father of Rohan"<<endl;
    }
};
class child:public parent {
    public:
    void display() {
        cout<<"I am the son of Naman"<<endl;
    }
};
```

```
int main() {
    child c;
    parent *ptr;
    ptr=&c;
    ptr->display();
    return 0;
}
```

Output:

```
I am the father of Rohan
```

```
#include <iostream>
using namespace std;
class parent {
    public:
    virtual void display() {
        cout<<"I am the father of Rohan"<<endl;
    }
};
class child:public parent {
    public:
    void display() {
        cout<<"I am the son of Naman"<<endl;
    }
};
```

```
int main() {
    child c;
    parent *ptr;
    ptr=&c;
    ptr->display();
    return 0;
}
```

Output:

```
I am the son of Naman
```

Function Overriding

Function overriding occurs when a function of the base class is given a new definition in a derived class. At that time, we can say the base function has been overridden.

```
#include<iostream>
using namespace std;
class parent{
    public:
    void msg(){
        cout<<"I am father of Rohan";
    }
};
class child:public parent{
    public:
    void msg(){
        cout<<"I am son of Naman";
    }
};
```

```
int main()
{
    child c;
    c.msg();
    return 0;
}
```

Output: *I am son of Naman*

Child class data will overriding the parent class

If you want to access both parent class and child class data then you have to call parent class method in child class.

```
#include<iostream>
using namespace std;
class parent{
    public:
    void msg(){
        cout<<"I am father of Rohan";
    }
};
class child:public parent{
    public:
    void msg(){
        cout<<"I am son of Naman\n";
        parent::msg();
    }
};
```

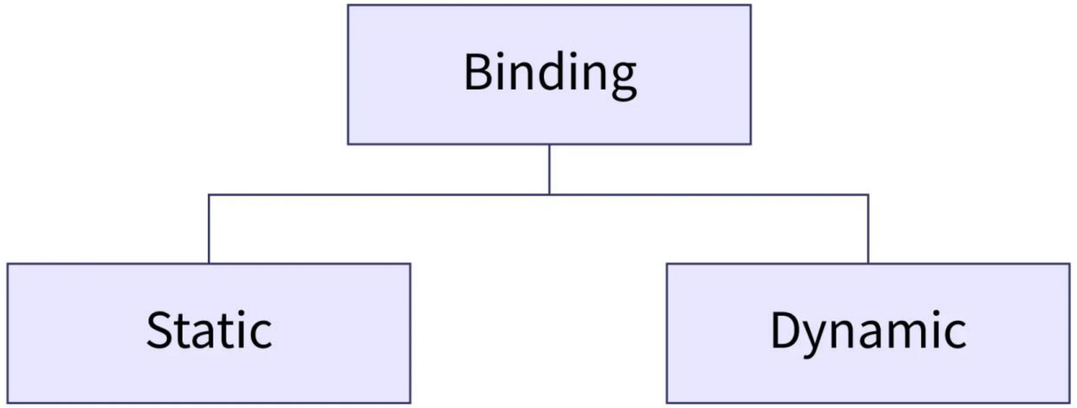
```
int main()
{
    child c;
    c.msg();
    return 0;
}
```

Output:

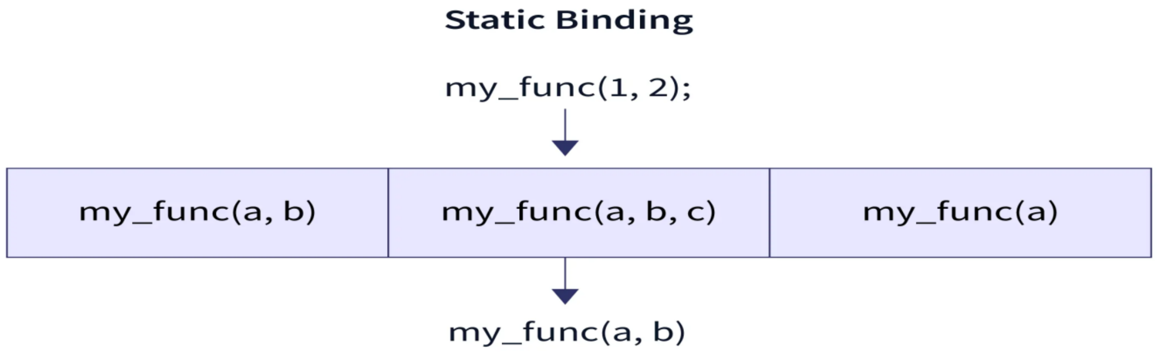
```
I am son of Naman
I am father of Rohan
```

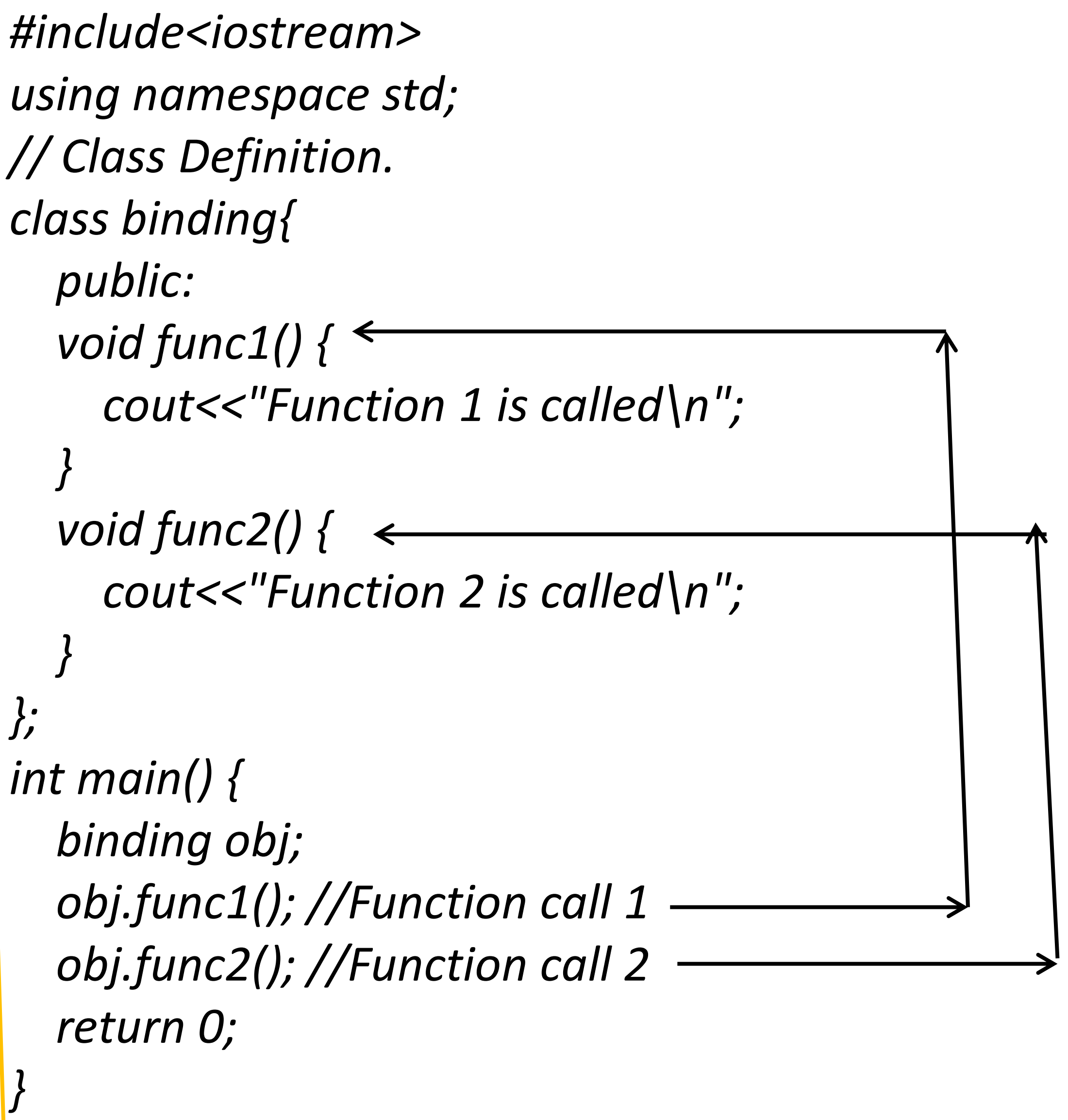
BINDING

- Binding helps in linking a bridge between a function call and its corresponding function definition.
- When we create a function, we have two crucial things:-
  - A function definition - defines a procedure to execute.
  - A function call - invokes the respective function for implementation.
- Now both the function definition and function calls are stored in the memory at separate addresses. And our program can have more than one function for its smooth operation. Hence, we need a technique to match the appropriate function call with its definition.
- The process of matching a specific function call to its respective function definition is known as binding.



- If there was no way of binding in C++, the two calls & definitions could've mixed up. Hence, binding is needed to link a function call with its function definition accurately.
- When this binding occurs at Compile time, we call it a Static Binding, and when this binding occurs at Runtime, we call it a Dynamic Binding.
- there are two ways by which static binding can be achieved: function overloading & operator overloading. Overloading.



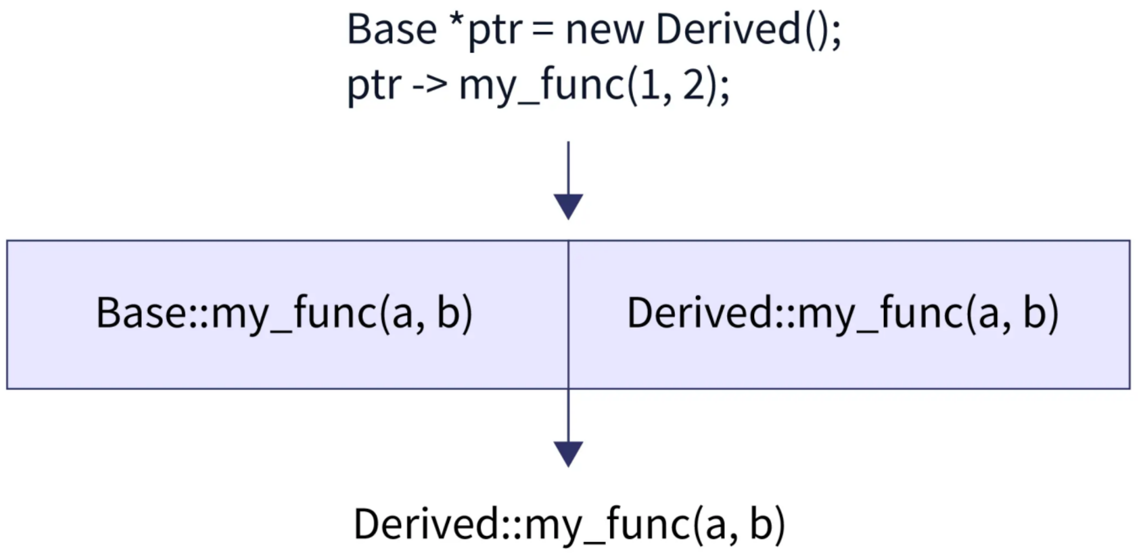


```
#include<iostream>
using namespace std;
// Class Definition.
class binding{
   public:
   void func1() {
      cout<<"Function 1 is called\n";
   }
   void func2() {
      cout<<"Function 2 is called\n";
   }
};
int main() {
   binding obj;
   obj.func1(); //Function call 1
   obj.func2(); //Function call 2
   return 0;
}
```

Dynamic Binding

- Binding at **runtime** is known as **dynamic binding**.
- There are instances in our program when the compiler cannot get all the information at compile time to resolve a function call. These function calls are linked at runtime. Such a process of binding is called dynamic binding or late binding.
- In C++, it is executed using virtual functions. A virtual function is a member function in the base class that is overridden (re-defined) by its derived class(es). When declaring in the base class, we use the keyword virtual to distinguish it.
- It is very flexible as one function is used to handle different objects.
- It also helps in reducing the overall size & making our program readable.
- However, since the linkage takes place during runtime, it can make the program slow.

Dynamic Binding



Basis of Comparison	Static Binding	Dynamic Binding
Event Occurrence	During compile time only. It is by default in C++.	During the run time.
Alternate Name(s)	Early Binding, Compile-time Binding	Late Binding, Runtime Binding
Information Available	All the information to resolve function calls is available simultaneously.	The compiler cannot determine all the information at compile time.
Advantage	It is efficient while executing as all the information is available before runtime.	It is flexible as different types of objects are handled at runtime by a single member function.
Speed	It is faster.	It is slower.
Example	Function overloading or operator overloading in C++.	Virtual functions in C++.

```
#include<iostream>
using namespace std;
class parent {
    public:
    virtual void display() {
        cout << "I am father of Rohan\n";
    }
};
class child: public parent {
    public:
    void display() {
        cout << "I am son of Naman\n";
    }
};
```

```
int main() {
    parent p;   // Object of parent class.
    child c;    // Object of child class.
    parent *ptr = &c;
    // Virtual function, binded at runtime.
    ptr->display();
    // Pointer of base class assigned address of its class.
    ptr = &p;
    // Virtual function, binded at runtime.
    ptr->display();
    return 0;
}
```

Output:

```
I am son of Naman
I am father of Rohan
```

Thank You