```cpp
public:
    // Override pure virtual function
    void makeSound() override {
        cout << "Cat meows" << endl;
    }
};

int main() {
    Animal* animal1 = new Dog(); // Creating a Dog object
    Animal* animal2 = new Cat(); // Creating a Cat object

    // Calling pure virtual function on Dog object
    animal1->makeSound(); // Output: Dog barks

    // Calling pure virtual function on Cat object
    animal2->makeSound(); // Output: Cat meows

    return 0;
}
```

In this example:

- We have an abstract base class Animal with a pure virtual function makeSound().
- There are two derived classes Dog and Cat, each overriding the makeSound() function with their specific implementation.
- In the main() function, we create objects of type Dog and Cat using base class pointers.
- When we call the makeSound() function on these objects, the appropriate version of the function is invoked based on the actual type of the object, demonstrating polymorphic behavior through dynamic dispatch.

Abstract Classes and Interfaces:

- **Abstract Class:** A class that contains at least one pure virtual function. It cannot be instantiated and serves as a blueprint for derived classes to implement common behavior while allowing specific implementations for their own unique features.
- **Interface:** C++ does not have a built-in concept of interfaces like Java or C#. However, interfaces can be simulated using abstract classes. An interface in C++ is an abstract class that has only pure virtual functions and no data members or non-virtual member functions. This ensures that the derived classes implement the specific methods defined by the interface.

Differences between Abstract Classes and Interfaces:

Feature	Abstract Classes	Interfaces
Definition	A class containing at least one pure virtual function	An abstract class containing only pure virtual functions

Purpose	To provide a common base class with some implementation and some methods to be overridden by derived classes	To define a contract that derived classes must follow
Implementation	Can contain some implementation (non-pure virtual functions) and member variables	Contains only pure virtual functions and no member variables
Instantiation	Cannot be instantiated directly	Cannot be instantiated directly
Inheritance	Derived classes can inherit only one abstract class (single inheritance)	A class can implement multiple interfaces (multiple inheritance)
Use Case	Use when you need a base class with some common behavior	Use when you need to define a clear contract for behavior without any implementation details

3.4 Virtual Destructors

In C++, when dealing with polymorphism and inheritance, it is often necessary to use virtual destructors to ensure that the proper destructors are called for objects of derived classes. A virtual destructor is a destructor declared in a base class using the virtual keyword, and it ensures that the destructors of both the base and derived classes are called in the correct order when deleting an object through a pointer to the base class. Virtual destructors become essential to ensure proper cleanup of resources allocated by derived classes.

Declaration and Syntax:

- The virtual keyword is used in the base class destructor to make it virtual.

```
class Base {
public:
    virtual ~Base() {
        // Virtual destructor
    }
};
```

- The derived class destructor overrides the base class destructor and provides its own implementation.

```
class Derived : public Base {
public:
    ~Derived() override {
        // Derived class destructor
    }
};
```

Purpose and Benefits:

1. Proper Destruction Order: Virtual destructors ensure that the destructors are called in the correct order when deleting objects through base class pointers, preventing memory leaks and undefined behavior.
2. Polymorphic Deletion: Virtual destructors enable the polymorphic deletion of objects, allowing for the correct cleanup of resources allocated by derived classes.
3. Prevents Memory Leaks: Helps avoid memory leaks by ensuring derived class destructors are called.

Example:

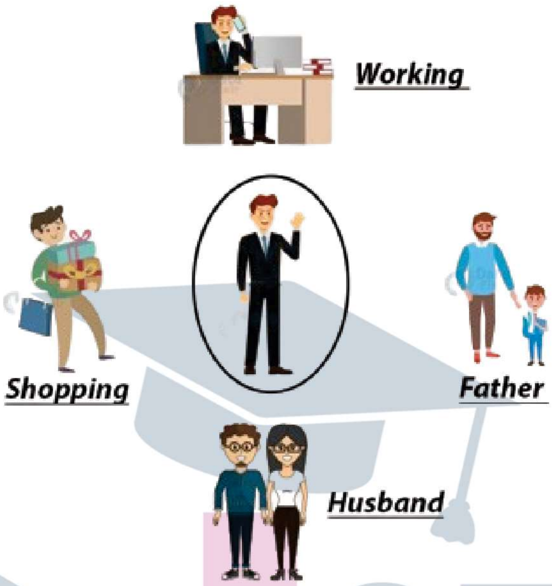
```
#include <iostream>
using namespace std;

// Base class with a virtual destructor
class Base {
public:
    virtual ~Base() {
        cout << "Base class destructor" << endl;
    }
};

// Derived class with its own destructor
class Derived : public Base {
public:
    ~Derived() override {
        cout << "Derived class destructor" << endl;
    }
};

int main() {
    Base* basePtr = new Derived(); // Creating a Derived object through a
Base pointer
    delete basePtr; // Deleting through a base class pointer

    return 0;
}
```

3.5 Polymorphism

Definition

- Polymorphism is derived from the Greek words "poly" (many) and "morphos" (forms).
- Polymorphism refers to the ability of objects to take on different forms or behaviors based on their context.

- In OOP, polymorphism refers to the ability of objects of different classes to be treated as objects of a common superclass.
- It allows a single interface (method or function) to represent multiple implementations.
- It allows a single function or operator to exhibit different behaviors based on the context in which it is called.
- Example: A man acts a father, husband, son, employee and many more.



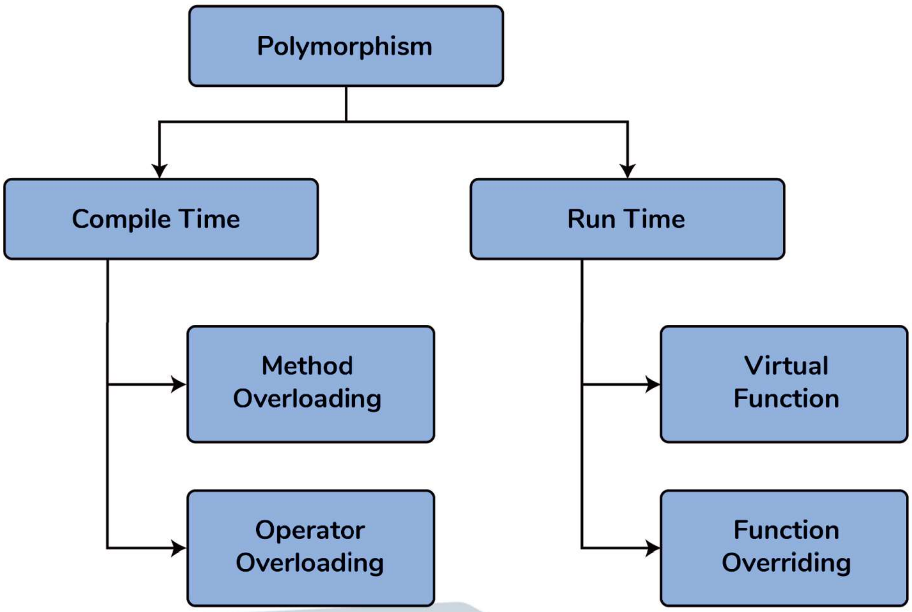
Benefits

1. Code Reusability: Polymorphism allows the same code to be reused with different objects, reducing duplication and improving maintainability.
2. Simplification: Polymorphism simplifies code maintenance and enhances readability by promoting a more modular and organized code structure.
3. Flexibility and Extensibility: It provides a flexible way to add new functionality to existing code by extending existing classes.
4. Encapsulation: Polymorphism promotes encapsulation by abstracting away the implementation details of objects and focusing on their behavior through a common interface.

Types of Polymorphism

Polymorphism can be of two types:

1. **Compile Time Polymorphism (Early / Static Binding)**
    - Achieved through method overloading and operator overloading.
    - Decisions about method calls are made at compile time.
    - Determined by the number and types of arguments and return type.
2. **Run Time Polymorphism (Late / Dynamic Binding)**
    - Achieved through virtual functions or method overriding.
    - Decisions about method calls are made at runtime.
    - Facilitated by pointers or references to base class objects.



Compile-time and Run-time

- **Compile Time (Early/Static):**
  - This is the phase when the **source code** written in programming language is being **converted into executable code** by a compiler.
  - During compile time, the compiler checks for **syntax errors** (like missing semicolons or mismatched brackets) and **semantic errors** (such as using a variable that hasn't been declared).
  - The compiler will not create an executable file until all such errors are resolved.
- **Run Time (Late/Dynamic):**
  - Run time refers to the period when the **executable code is actually running** on computer.
  - It's the phase where the program interacts with inputs, performs calculations, and may encounter **runtime errors**. These errors occur during execution and can include issues like division by zero or accessing an array out of bound.

A. Compile-time Polymorphism (Static Polymorphism):

- Polymorphism that is resolved at compile time when the **source code** is being **converted into executable code** by compiler.
- It is achieved through method overloading and operator overloading, where the compiler selects the appropriate function or operator based on the arguments and context at compile time.
- Also known as **static or early binding**, this type of polymorphism is achieved through method overloading and operator overloading.

Method Overloading:

- Definition: Method overloading is a form of compile-time polymorphism where multiple methods in the same class have the same name but differ in the number or type of their parameters.
- Methods can be overloaded by changing the number or type of arguments.
- It provides flexibility and clarity in code by allowing multiple functions with similar functionality to be grouped under the same name.
- Example:

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

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

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

    double add(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculator calc;
    cout << calc.add(5, 7) << endl;         // Output: 12
    cout << calc.add(5, 7, 3) << endl;      // Output: 15
    cout << calc.add(3.5, 2.5) << endl;     // Output: 6
    return 0;
}
```

Operator Overloading:

- Definition: Operator overloading is a form of compile-time polymorphism where operators are overloaded to work with user-defined data types.
- It allows defining custom behavior for operators based on the data types involved.
- Example:

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

class Complex {
private:
    int real, imag;
```

```
public:
    // Constructor to initialize real and imaginary parts, default values are 0
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}

    // Overloading the + operator to add two complex numbers
    Complex operator+(Complex const& obj) {
        // Adding real parts and imaginary parts separately
        return Complex(real + obj.real, imag + obj.imag);
    }

    // Function to print the complex number in the format "real + imagi"
    void print() { cout << real << " + " << imag << "i" << endl; }
};

int main() {
    // Creating two complex numbers
    Complex c1(10, 5), c2(2, 4);

    // Adding two complex numbers using overloaded + operator
    Complex c3 = c1 + c2;

    // Printing the result
    cout << "Result of addition: ";
    c3.print();

    return 0;
}
```

Output:

```
Result of addition: 12 + 9i
```

B. Run-time Polymorphism (Dynamic Polymorphism):

- Polymorphism that is resolved at runtime when the **executable code is actually running** on computer.
- It is achieved through method overriding or virtual functions, where the appropriate function to call is determined dynamically based on the actual object type at runtime.
- Also known as **dynamic or late binding**, this occurs during program execution.
- Achieved through virtual functions (using the virtual keyword).
- Allows a base class pointer to invoke derived class methods.

Virtual Functions:

- Definition: Virtual functions are used in run-time polymorphism to enable dynamic method binding. They are member functions which are declared in the base class with the virtual keyword and can be overridden in derived classes.
- They enable dynamic binding of function calls, allowing the correct function to be called at runtime based on the type of object.
- Example:

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

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

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

int main() {
    Shape* shape = new Circle();
    shape->draw();  // Output: Drawing Circle

    return 0;
}
```

Method Overriding:

- Definition: Method overriding is a form of run-time polymorphism where a method in a base class is redefined in a derived class. The method in the derived class must have the same signature (name and parameters) as the one in the base class.
- It allows derived classes to provide a specific implementation of a method defined in the base class, promoting flexibility and extensibility.
- Example:

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

class Animal {
public:
    virtual void sound() {
        cout << "Animal makes a sound" << endl;
```

```
    }
};

class Dog : public Animal {
public:
    void sound() override {
        cout << "Dog barks" << endl;
    }
};

int main() {
    Animal* animal = new Dog();
    animal->sound();  // Output: Dog barks

    return 0;
}
```

Assignment

1. Define Early Binding in C++.
2. Define Late Binding in C++.
3. What is Virtual Function.
4. Explain Pure Virtual Functions.
5. Explain Abstract Classes.
6. What are Virtual Destructures in C++.