Inheritance and Polymorphism in C++

1 What is Inheritance?

Inheritance allows a class (derived) to reuse the code from another class (base). It promotes code reuse and polymorphism.

2 Basic Syntax

```cpp
class Base {
public:
    void sayHello() {
        std::cout << "Hello from Base\n";
    }
};

class Derived : public Base {
    // Inherits from Base
};
```

Protected members of the base class are accessible in the derived class, while private members are not.

3 Access Specifiers in Inheritance

Inheritance Type	`public` in Base	`protected` in Base	`private` in Base
`public`	public	protected	inaccessible
`protected`	protected	protected	inaccessible
`private`	private	private	inaccessible

4 Constructors and Destructors

```cpp
class Animal {
public:
    Animal() {
        std::cout << "Animal created\n";
    }
    ~Animal() {
        std::cout << "Animal destroyed\n";
    }
};

class Dog : public Animal {
public:
    Dog() {
        std::cout << "Dog created\n";
    }
    ~Dog() {
        std::cout << "Dog destroyed\n";
    }
};
```

Output when creating a `Dog` object: Note the order in which constructors and destructors are called.

```
Animal created
Dog created
Dog destroyed
Animal destroyed
```

Demonstration of inheritance of a Student class from Person class.

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

class Person {
protected:
  string name;
  int age;
public:
  Person(string name, int age) : name(name), age(age) {}
  string getName() { return name; }
  int getAge() { return age; }
 };

class Student : public Person {
private:
  string major;
public:
  Student(string name, int age, string major) : Person(name, age), major(major) {};
  string getMajor() { return major; }
 };

 int main() {
  Student student("John Doe", 20, "Computer Science");
  cout << "Name: " << student.getName() << endl; // Inhertited from Person
  cout << "Age: " << student.getAge() << endl; // Inherited from Person
  cout << "Major: " << student.getMajor() << endl;
 }
```

5 Virtual Functions and Dynamic Dispatch

A virtual function is a member function in a base class that you expect to be overridden in derived classes. It allows dynamic dispatch, meaning the function that gets called is determined at runtime, based on the actual type of the object, not the type of the pointer/reference.

5.1 Without Virtual

```
class Base {
public:
    void speak() {
        std::cout << "Base speaking\n";
    }
};

class Derived : public Base {
public:
    void speak() {
        std::cout << "Derived speaking\n";
    }
};

Base* b = new Derived();
b->speak();  // Outputs: Base speaking
```

5.2 With Virtual

```
class Base {
public:
    virtual void speak() {
        std::cout << "Base speaking\n";
    }
};

class Derived : public Base {
public:
    void speak() override {
        std::cout << "Derived speaking\n";
    }
};

Base* b = new Derived();
b->speak();  // Outputs: Derived speaking
```

Why use virtual destructors?

```
class Base {
public:
    virtual ~Base() {
        std::cout << "Base destroyed\n";
    }
};

class Derived : public Base {
public:
    ~Derived() {
        std::cout << "Derived destroyed\n";
    }
};

Base* b = new Derived();
delete b; // Calls Derived and Base destructors
```

6 Dynamic Memory + Rule of Three

If a class uses dynamic memory, it must:

- Define a destructor
- Define a copy constructor
- Define a copy assignment operator

```
class Base {
protected:
    char* name;
public:
    Base(const char* n) {
        name = new char[strlen(n)+1];
        strcpy(name, n);
    }

    virtual ~Base() {
        delete[] name;
    }

    // Copy constructor and assignment omitted for brevity
};
```

7 Polymorphism in C++

Polymorphism means “many forms”. In C++, it refers to the ability of different classes to be treated as instances of the same base class, particularly when using pointers or references. There are two main types:

- **Compile-time polymorphism** (e.g., function overloading, operator overloading)
- **Run-time polymorphism** (using `virtual` functions)

7.1 Compile-Time Polymorphism

Achieved through **function overloading** or **operator overloading**.

```
class Printer {
public:
    void print(int i) {
        std::cout << "Integer: " << i << "\n";
    }
    void print(const std::string& s) {
        std::cout << "String: " << s << "\n";
    }
};
```

7.2 Run-Time Polymorphism

Achieved using **virtual functions**. The function that gets invoked depends on the actual type of the object pointed to.

```
class Shape {
public:
    virtual double getArea() const = 0;  // Pure virtual function
    virtual ~Shape() {}
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}
    double getArea() const override {
        return 3.14159 * radius * radius;
    }
};

class Rectangle : public Shape {
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double getArea() const override {
        return width * height;
    }
};
```

7.3 Example: Using Base Pointers

```
Shape* shapes[2];
shapes[0] = new Circle(2.0);
shapes[1] = new Rectangle(3.0, 4.0);
```

```cpp
for (int i = 0; i < 2; ++i) {
    std::cout << "Area: " << shapes[i]->getArea() << "\n";
    delete shapes[i];
}
```

Output:

```
Area: 12.56636
Area: 12
```

7.4 Notes

- To enable polymorphism, use `virtual` in base class.
- A pure virtual function (declared with `= 0`) makes a class abstract.
- Always declare destructors `virtual` in base classes when using polymorphism.

8 Abstract Classes in C++

An **abstract class** in C++ is a class that cannot be instantiated on its own. It serves as a blueprint for other classes and typically contains at least one **pure virtual function**.

8.1 Pure Virtual Functions

A pure virtual function is declared by assigning `= 0` in its declaration:

```cpp
class Shape {
public:
    virtual double getArea() const = 0; // pure virtual function
};
```

Any class with at least one pure virtual function is an **abstract class**.

8.2 Purpose of Abstract Classes

- Define a common interface for derived classes.
- Enforce implementation of certain methods in derived classes.
- Enable polymorphism with base pointers or references.

8.3 Example: Abstract Base Class

```cpp
class Shape {
public:
    virtual double getArea() const = 0; // pure virtual
    virtual void draw() const = 0;
    virtual ~Shape() {}
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}
    double getArea() const override {
        return 3.14159 * radius * radius;
    }
    void draw() const override {
        std::cout << "Drawing circle\n";
    }
};

class Rectangle : public Shape {
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double getArea() const override {
        return width * height;
    }
    void draw() const override {
        std::cout << "Drawing rectangle\n";
    }
};
```

8.4 Usage with Polymorphism

```cpp
Shape* s1 = new Circle(2.0);
Shape* s2 = new Rectangle(3.0, 4.0);

s1->draw();   // Drawing circle
s2->draw();   // Drawing rectangle

delete s1;
delete s2;
```

8.5 Key Points

- Abstract classes define interfaces, not implementations.
- Cannot create instances of abstract classes.
- Derived classes must implement all pure virtual functions, or they remain abstract.
- Use abstract base classes to enable consistent APIs and polymorphism.

9 Multiple Inheritance (Overview)

```cpp
class A { };
class B { };
class C : public A, public B { };
```

Be careful with ambiguity and name conflicts.