CHAPTER-6

Virtual function, Polymorphism and miscellaneous c++ features

Before starting virtual function lets check why it is needed?

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

class Animal {
public:
    void sound() {
        cout << "Animal makes sound" << endl;
    }
};

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

int main() {
    Dog d;
    d.sound();  // Output: Dog barks

    Animal a;
    a.sound();  // Output: Animal makes sound

    return 0;
}
```

It is also called Early Binding (also called **Static Binding**) means the **compiler already knows** at **compile time** which function to call.

Prepared by Ayush Lamsal ||
layush788@gmail.com

Without virtual function

dynamic_binding.cpp > ...

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

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

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

int main() {
    Base* ptr;       // Pointer of base class
    Derived d;       // Object of derived class
    ptr = &d;        // Base class pointer points to derived class object

    ptr->show();     // Output: Base class show()
    // Even though ptr points to a Derived object,
    // it calls Base::show() because 'show' is not declared virtual.

    return 0;
}
```

Prepared by Ayush Lamsal ||
layush788@gmail.com

Using a virtual keyboard

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

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

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

int main() {
    Base* ptr;       // Pointer of base class
    Derived d;       // Object of derived class
    ptr = &d;        // Base class pointer points to derived class object

    ptr->show();     // Output: Base class show()
    // Even though ptr points to a Derived object,
    // it calls Base::show() because 'show' is not declared virtual.

    return 0;
}
```

// Output: Derived class show()
// **'show' is declared as virtual in the base class.**
// So this is an example of late binding (dynamic binding).
// **The decision of which 'show' to call is made at runtime,**
// depending on the actual object the pointer points to.

What is virtual function

- A virtual function is a member function that is:
  - *Declared in the base class using the keyword virtual, and*
  - *Overridden in the derived class with the same function name and signature.*
- When virtual functions are used, the same function call (e.g., ptr->show()) can execute **different versions of the function depending on the type of object the pointer refers to** — this is called runtime polymorphism or late binding.
- Virtual function cannot be static and also cannot be a friend function or another class.
- Virtual function should be accessed using the pointer or reference of base class to achieve the run time polymorphism
- The prototype of virtual function should be same in base class as well as in derived class.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Example of virtual function

We have done previously

Prepared by Ayush Lamsal | |
layush788@gmail.com

What is polymorphism

Polymorphism means **“many forms.”** It means that the **same name** (like a function or operator) can perform **different actions** depending on the context.

Example:

- A function named draw() can draw different shapes — like circles, rectangles, or triangles — depending on the object calling it.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Types of polymorphism

Compile-time Polymorphism (Static Binding)

- The function to be called is determined at compile time.
- Achieved by:
  - Function Overloading (same function name, different parameters)
  - Operator Overloading (customizing operators for user-defined types)

Run-time Polymorphism (Dynamic Binding)

- The function to be called is determined at runtime based on the actual object type.

Achieved by:

- Virtual functions

Prepared by Ayush Lamsal | |
layush788@gmail.com

Example of runtime polymorphism(method overriding)

- Run-time polymorphism allows a **program to decide which function to call at runtime rather than compile time. It is a core feature of Object-Oriented Programming that supports flexible and reusable code.**

How it works:

- Achieved by using **virtual functions in a base class**.
- When a **base class pointer or reference points to a derived class object**, the derived class's overridden function is called.
- The decision **of which function to execute is made dynamically during program** execution (late binding).

Prepared by Ayush Lamsal | |
layush788@gmail.com

Example of runtime polymorphism

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

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

class Derived : public Base {
public:
    void show() override {             // Overrides Base's show()
        cout << "Derived class show()" << endl;
    }
};

int main() {
    Base* ptr;                        // Pointer of base class type
    Derived d;                       // Object of derived class
    ptr = &d;                        // Base class pointer points to derived class object

    ptr->show();                    // Calls Derived::show() at runtime
    // Because 'show' is virtual, the call is resolved dynamically (runtime polymorphism)
    // The actual function called depends on the type of the object pointed to by ptr

    return 0;
}

```

layush788@gmail.com

Difference between normal member function accessed with pointer and virtual member function accessed with pointer

- In C++, member functions of a class can be either **normal** or **virtual**. When these functions are accessed through a **pointer**, their behavior differs based on how the function is bound to the call.
- **Normal Member Function**
- A normal (non-virtual) function is **statically bound**.
- This means the function to be called is decided at **compile time**, based on the **type of the pointer**, not the actual object it points to.
- It does **not support polymorphism**, so the **derived class function will not be called if accessed through a base class pointer.**

Prepared by Ayush Lamsal | |
layush788@gmail.com

Difference between normal member function accessed with pointer and virtual member function accessed with pointer

Virtual Member Function

- A virtual function is **dynamically bound**.
- This means the function to be called is decided at **runtime**, based on the **actual object** the pointer points to.
- It supports **runtime polymorphism**, allowing derived class functions to override base class functions.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Example

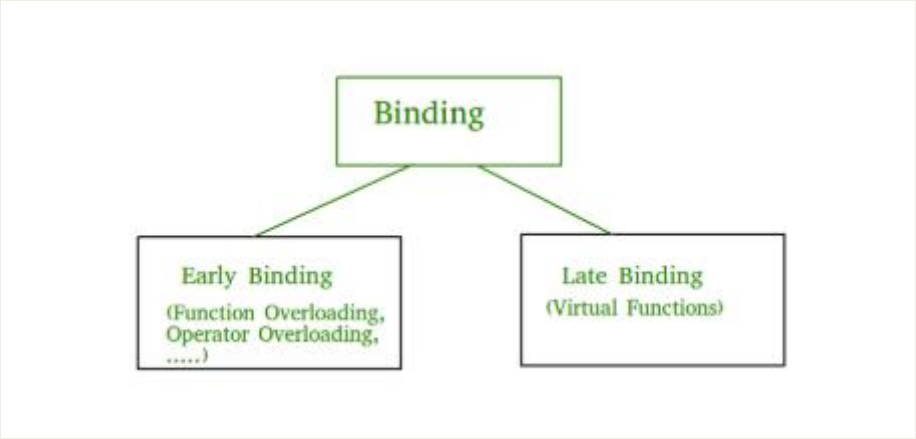
We have already covered examples of both normal member functions (accessed without the virtual keyword) and virtual member functions earlier.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Early binding and late binding in c++

- Binding refers to the process of connecting a function call to the actual code (function definition) that will be executed.
- In simpler terms, Binding decides which function gets called when your program runs.

There are two types of binding in C++:



Prepared by Ayush Lamsal ||
layush788@gmail.com

Early binding and late binding in c++

Early Binding (also called Static Binding)

- The function call is **resolved at compile time**.
- The **compiler** decides which function to call based on the **type of the pointer or object**.
- Used for **normal (non-virtual)** functions.

Late Binding (also called Dynamic Binding)

- The function call is resolved at runtime.
- The decision is made based on the actual object type, not the pointer type.
- Happens only when the function is marked with the virtual keyword.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Example of Early binding and late binding

We have already covered this example

Prepared by Ayush Lamsal | |
layush788@gmail.com

Pure virtual Function

A pure virtual function is a function declared in a base class that must be overridden by any derived class.

It has no implementation in the base class and is specified by assigning = 0 in its declaration.

Syntax:

```
virtual returnType functionName(parameters) = 0;
```

It means the base class **forces derived classes to implement** this function.

When a class contains at least one pure virtual function, it becomes an **abstract class**.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Example

We have already covered in previous chapter

Prepared by Ayush Lamsal | |
layush788@gmail.com

What is abstract class

- An abstract class is a class that contains at least one pure virtual function (a function declared with = 0 and without implementation). Because of this, an abstract class cannot be instantiated directly, **meaning you cannot create objects of an abstract class.**
- The purpose of an abstract class is to provide **a base class that defines a common interface and possibly some shared behavior**, while forcing derived classes to **implement the pure virtual functions**.
- Abstract classes **are used to achieve runtime polymorphism** and enforce a contract for derived classes.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Example

We covered the example of abstract class in previous chapter

Prepared by Ayush Lamsal | |
layush788@gmail.com

Virtual Destructor

A virtual destructor is a destructor **declared with the virtual keyword** in a base class to ensure **that the correct destructor is called when deleting an object through a base class** pointer.

Why use a virtual destructor?

- When you delete an object through a **pointer to a base class**, and the base class destructor is **not virtual**, only the base class destructor is called.
- This causes **resource leaks** or incomplete destruction if the derived class has its own destructor.
- Declaring the base class destructor as **virtual** ensures the **derived class destructor is called first**, followed by the base class destructor — enabling proper cleanup.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Virtual Destructor (Example)

virtual_destructor.cpp > Derived > ~Derived()

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

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

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived destructor called" << endl;
    }
};

int main() {
    Base* ptr = new Derived();
    delete ptr;
    // Output:
    // Derived destructor called
    // Base destructor called
    return 0;
}

```

Prepared by Ayush Lamsal ||
layush788@gmail.com

Virtual Destructor (Example)

virtual_destructor.cpp > ...

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

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

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived destructor called" << endl;
    }
};

int main() {
    Base* ptr = new Derived();
    // Here, 'new Derived()' creates an object of Derived on the **heap** (dynamic memory).
    // This is different from 'Derived d;' which creates an object on the **stack** (automatic memory).
    // Deleting objects created with 'new' using 'delete' is correct and necessary to free heap memory.
    // However, deleting an object created on the stack (like 'Derived d;') using 'delete' causes undefined behavior.

    delete ptr;
    // Output:
    // Derived destructor called
    // Base destructor called
    return 0;
}
```

What without virtual keyword in destructor?

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

class Base {
public:
    ~Base() {  // Non-virtual destructor
        cout << "Base destructor called" << endl;
    }
};

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived destructor called" << endl;
    }
};

int main() {
    Base* ptr = new Derived();
    // 'new Derived()' creates a Derived object on the heap.

    delete ptr;
    // Output:
    // Base destructor called
    // Derived destructor is not called so meory leaks
    return 0;
}

```

layusn788@gmail.com

What without virtual keyword in destructor?

Without a virtual destructor in the base class, when you delete a pointer of base class type, only the base class destructor is called, regardless of the actual type of the object it points to.

This means the derived class destructor is skipped, which can cause resource leaks or undefined behavior.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Static member function in c++

In C++, a **static member function** is a special kind of class function that belongs to the **class itself**, rather than to any particular object of the class.

This means that static member functions are associated with the class as a whole and not with any individual instance or object.

1. Declaration and Definition

A static member function is declared by using the keyword static inside the class definition, just like static data members.

The function is defined similarly to other member functions, but it has special properties related to how it accesses class members and how it is called.

2. Calling Static Member Functions

Unlike regular member functions, which require an object to be called, static member functions can be invoked without creating an object of the class.

They can be called directly using the class name and the scope resolution operator ::

Prepared by Ayush Lamsal | |
layush788@gmail.com

Example of static member function

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

class Message {
public:
    static void printMsg() {
        cout << "Hello from static function!" << endl;
    }
};

int main() {
    // Call the static function using class name (no object needed)
    Message::printMsg();

    return 0;
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

Static member function in c++

stastic.cpp > ...

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

class MyClass {
    int x = 10;              // Non-static: unique to each object
    static int y;            // Static: shared among all objects

public:
    // Normal (non-static) method can access both static and non-static members
    void showBoth() {
        cout << "x = " << x << ", y = " << y << endl;
    }

    // Static method: can only access static members (no 'this' pointer)
    static void showStatic() {
        cout << "y = " << y << endl;
        // cout << x;  Error: Cannot access non-static member from static context
    }
};

// Static member initialization outside the class
int MyClass::y = 20; // This is required

int main() {
    MyClass obj;
    obj.showBoth();        // Prints x = 10, y = 20
    MyClass::showStatic(); // prints y = 20
    return 0;
}

```

Prepared by Ayush Lamsal ||
layush788@gmail.com

Static data member in c++ (Revision)

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

class MyClass {
public:
    static int count;  // Static data member

    static void showCount() {
        cout << "Count = " << count << endl;
    }
};

// Define and initialize static member outside the class
int MyClass::count = 100;

int main() {
    // Access static data member without creating any object
    cout << "Accessing static member directly: " << MyClass::count << endl;

    // Call static member function without creating any object
    MyClass::showCount();

    return 0;
}
```

Prepared by Ayush Lamsal ||
layush788@gmail.com

Another example of Static data member in c++ (Revision)

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

class A {
public:
    static int x;  // Static data member declaration
};

// Definition and initialization of static data member outside the class
int A::x = 5;

class B {
public:
    void print() {
        // cout << x;    // ERROR: 'x' is not a member of class B
        cout << "Accessing A::x from class B: " << A::x << endl;  // Correct access
    }
};

int main() {
    B b;
    b.print();  // Output: Accessing A::x from class B: 5

    // Direct access from main without object of A
    cout << "Direct access to A::x in main: " << A::x << endl;

    return 0;
}

```

Prepared by Ayush Lamsal ||
layush788@gmail.com

Concrete class vs Abstract class example

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

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

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

int main() {
    // Shape s;  // Error: Cannot instantiate abstract class

    Circle c;   // Create object of concrete class
    c.draw();   // Output: Drawing Circle

    Shape* ptr = &c;
    ptr->draw();  // Calls Circle's draw() (polymorphism)

    return 0;
}
```

Concrete class vs Abstract class Theory

There are two main types of classes in object-oriented programming:

Concrete Class

- A **Concrete Class** is a regular class that provides **complete implementation** of all its member functions.
- You can **create objects** of a concrete class.
- It may or may not have virtual functions, but it **does not have any pure virtual functions**.
- Example: A class that implements all its methods and can be instantiated.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Concrete class vs Abstract class Theory

Abstract Class

- An Abstract Class is a class that contains at least one pure virtual function.

  A pure virtual function is declared like this:

  virtual void functionName() = 0;

- You cannot create objects of an abstract class directly.
- Abstract classes are meant to be base classes, providing a common interface or template for all derived classes.
- The derived classes must override and provide definitions for all pure virtual functions to become concrete (usable) classes.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Pointer to base class

- In C++, when one class is inherited from another, a pointer of the base class type can be used to point to an object of the derived class. This concept is known as a pointer to base class and the process is called upcasting.
- This is useful for achieving runtime polymorphism, especially when virtual functions are involved.
- ⚠ Without virtual functions, the base class version of the method will be called even if the pointer is pointing to a derived class object.

Prepared by Ayush Lamsal | |
layush788@gmail.com

Pointer to base class (Example)

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

class Animal {
public:
    virtual void speak() {
        cout << "Animal speaks" << endl;
    }
};

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

int main() {
    Dog d;
    Animal* ptr;        // Pointer to base class
    ptr = &d;           // Upcasting: Derived object assigned to base pointer

    ptr->speak();       // Output: Dog barks (due to virtual function)

    return 0;
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

This pointer

- The "**this pointer in C++**" is a special keyword available inside non-static member functions of a class.
- It holds the address of the current object that is invoking the member function.
- It is not used in static member functions because static functions do not belong to any object — they are associated with the class itself, not with any particular instance.
- This pointer is passed as a hidden argument to all non static member
- Following are the situation where the this pointer is used
  - *When local variable name is same as member variable name*
  - *To return reference to the calling object (Useful for method chaining)*

Prepared by Ayush Lamsal | |
layush788@gmail.com

When local variable name is same as member name

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

class MyClass {
    int x;  // Member variable

public:
    void setX(int x) {  // Parameter has the same name 'x'
        this->x = x;    // 'this->x' is the member variable, 'x' is the parameter
    }

    void show() {
        cout << "Value of x: " << x << endl;
    }
};

int main() {
    MyClass obj;
    obj.setX(10);  // Set x to 10
    obj.show();    // Output: Value of x: 10
    return 0;
}
```

Prepared by Ayush Lamsal ||
layush788@gmail.com

Note: Try this without this pointer

To return reference of the calling object

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

class MyClass {
public:
   //here MyClass& is a return type of a pointer as this method is returning the object of a class type MyClass
    MyClass& sayHello() {
        cout << "Hello ";
        return *this;  // Return current object by reference
        //it first print the hello world and return the same object which can be again used to call another method here sayworld()
    }

    MyClass& sayWorld() {
        cout << "World!" << endl;
        return *this;
    }
};

int main() {
    MyClass obj;
    obj.sayHello().sayWorld();  // Prints "Hello World!" using method chaining
    return 0;
}
```

Prepared by Ayush Lamsal ||
layush788@gmail.com

Important Assignment

1. List five difference between method overloading and overriding
2. What is the difference between abstract class and concrete class
3. What is the difference between Early Binding and late binding

Prepared by Ayush Lamsal | |
layush788@gmail.com

Thank You

Prepared by Ayush Lamsal ||
layush788@gmail.com