**LAB 11:TO ILLUSTRATE THE CONCEPT OF RUN TIME POLYMORPHISM IN C++.**

**OBJECTIVE:**

1.Understand the concept of polymorphism in object-oriented programming.

2.Learn the difference between compile-time and runtime polymorphism.

3.Implement runtime polymorphism using virtual functions in C++.

4.Create a practical example demonstrating the usage of runtime polymorphism to enhance code flexibility and maintainability.

**THEORY:** Object-oriented programming (OOP) is a paradigm that revolves around the concept of objects, which are instances of classes, and is renowned for its ability to model real-world entities in a more intuitive and organized manner. One of the core features of OOP is polymorphism, a term derived from the Greek words "poly" (many) and "morph" (form), which essentially means having multiple forms. Polymorphism in C++ is a powerful concept that contributes significantly to the flexibility and maintainability of code. In particular, runtime polymorphism is a fundamental aspect of polymorphism that allows different objects to respond to the same method call in a way that is appropriate for their individual types. In this laboratory exercise, we delve into the intricacies of runtime polymorphism in C++, aiming to provide a comprehensive understanding of its importance and practical implementation. It allows objects of different classes to be treated as objects of a common base class, enabling code to be written in a more generic and extensible manner.

**There are two primary types of polymorphism in C++:**

**1.Compile-Time Polymorphism (Static Binding):** This form of polymorphism is resolved during the compile-time, also known as static binding. It is achieved through function overloading and operator overloading. In compile-time polymorphism, the compiler determines the appropriate function or operator to call based on the function's signature or operator symbol.

**2.Runtime Polymorphism (Dynamic Binding):** Unlike compile-time polymorphism, runtime polymorphism is resolved during program execution, also known as dynamic binding. It is achieved through the use of virtual functions and is a hallmark feature of OOP. In runtime polymorphism, the correct function to be executed is determined at runtime based on the actual type of the object being referred to.

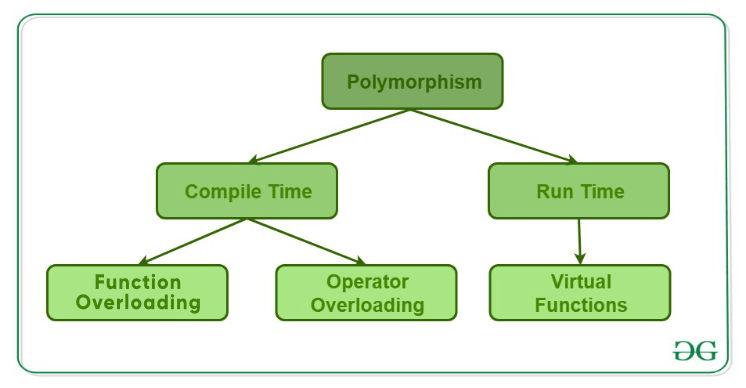


Fig:polymorphism

**SOURCE CODE:**

#include <iostream>

using namespace std;

class Game { //base class

public:

virtual void display() {

cout << "I love playing games" <<endl;

}

};

class Cricket : public Game { //derived class:cricket

public:

void display() override {

cout << "I love playing cricket." <<endl;

}

};

class Football: public Game { //derived class:football

public:

void display() override {

cout << "I love playing football." << endl;

}

};

int main() {

//create the objects of cricket and rectangle.

Cricket cricket;

Football football;

//pointers of base class type.

Game\* game1 = &cricket;

Game\* game2= &football;

//call display() using base clas pointer.

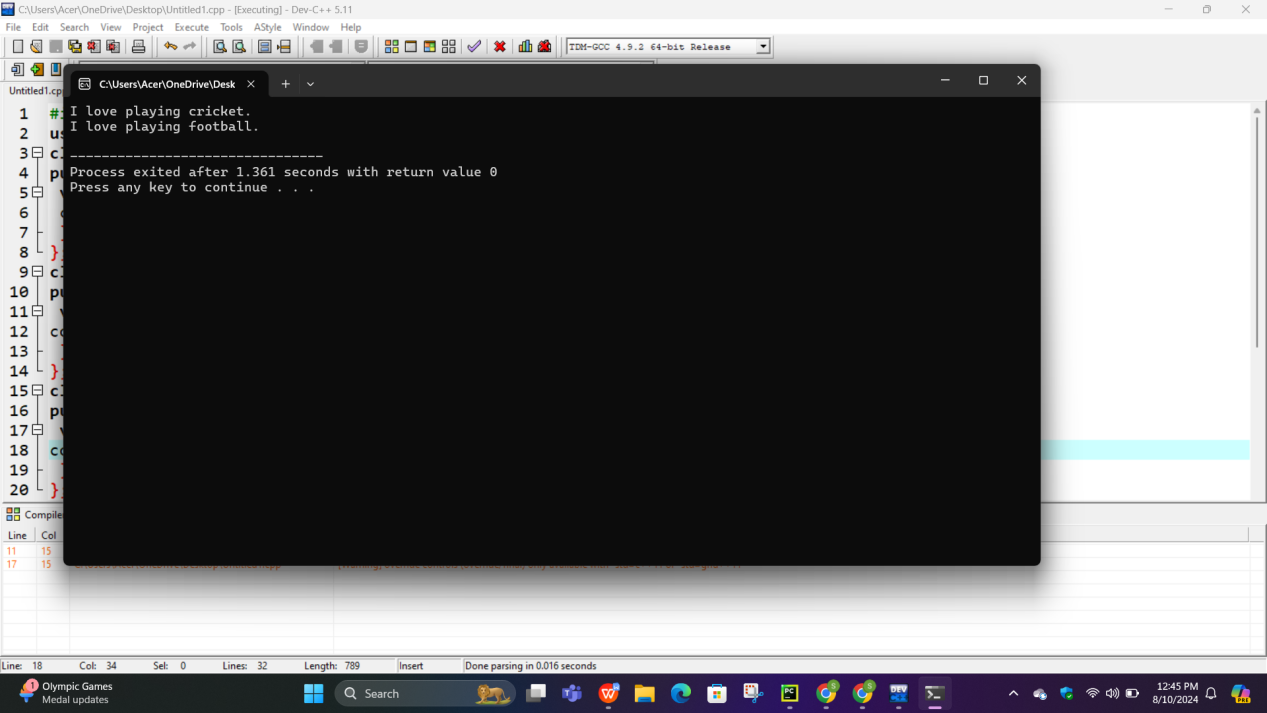
game1->display(); //call display()of cricket.

game2->display(); // call display()of football.

return 0;

}

**OUTPUT:**



**CONCLUSION:** In this lab, we successfully demonstrated the concept of runtime polymorphism in C++ using virtual functions. Runtime polymorphism, a key feature of object-oriented programming, enables objects of different classes to be handled uniformly through a common base class pointer. This dynamic binding, resolved during program execution, enhances code flexibility and maintainability. In practical terms, runtime polymorphism is a powerful tool for writing adaptable and maintainable C++ code. Understanding virtual functions and dynamic binding equips programmers with the ability to create efficient and extensible systems. This lab serves as a foundational step toward mastering these essential concepts in C++ development.