**LAB 6:TO ILLUSTRATE THE CONCEPT OF FUNCTION OVERLOADING IN C++.**

**OBJECTIVE:**

1.Understand the concept of function overloading in C++.

2.Learn how to create multiple functions with the same name but different parameters.

3.Practice defining overloaded functions to perform different tasks based on input arguments.

**THEORY:** Function overloading is a powerful feature in C++ that allows you to define multiple functions with the same name in the same scope, but with different parameter lists. This enables you to create more intuitive and flexible code by providing different ways to use a function depending on the arguments provided. Here's a detailed explanation of function overloading with suitable real-world examples to illustrate its importance and usefulness. Function overloading is a form of compile-time polymorphism It enables you to create multiple functions with the same name but different parameter lists, and the appropriate function to call is determined by the number and types of arguments during compilation. This means that at runtime, the correct function to execute is already known, resulting in efficient code execution.

**Benefits of Function Overloading:**

1.**Readability**: Function overloading enhances code readability by using the same name for logically related operations, making the code more intuitive and self-explanatory.

2.**Code Reusability**: Instead of creating distinct function names for similar operations, function overloading allows you to reuse function names with different parameter lists.

3.**Consistency**: It promotes consistency in naming conventions and function usage throughout the codebase, which makes maintenance easier.

4.**Flexibility**: Function overloading provides flexibility for users of your code, as they can choose the appropriate version of the function based on their specific needs.

**PROGRAM 1:WAP TO OVERLOAD THE FUNCTION SUM() THAT CAN ADD TWO INTEGERS TWO FLOATS AND TWO STRINGS.**

**//SOURCE CODE:**

#include<iostream>

using namespace std;

void sum(int a , int b){

cout<<"The sum of "<<a<< " and " <<b<< " is "<<a+b<<endl;

}

void sum(double a , double b){

cout<<"The sum of "<<a<< " and " <<b<< " is "<<a+b<<endl;

}

void sum(string s1,string s2){

cout<<s1<<s2<<endl;

}

int main(){

sum(2,4);

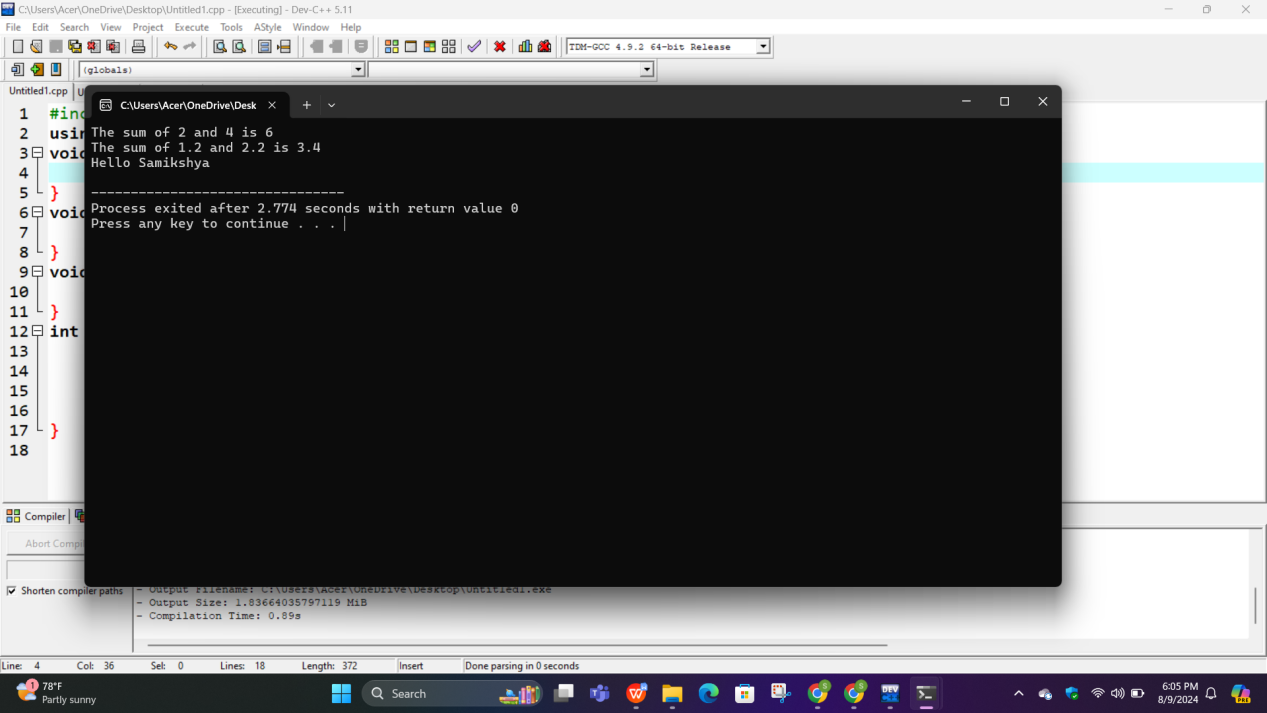
sum(1.2,2.2);

sum("Hello " , "Samikshya");

return 0;

}

**OUTPUT:**

****

**PROGRAM 2:WAP TO OVERLOAD CONSTRUCTOR USING ANY CLASS OF YOUR CHOICE.**

**//SOURCE CODE:**

#include<iostream>

using namespace std;

class Teacher{

string name;

int id;

public:

Teacher(){

name="Gyaneshwor";

id=23;

}

Teacher(string n , int i){

name=n;

id=i;

}

Teacher(Teacher &x){

name=x.name;

id=x.id;

}

void display(){

cout<<"Name:"<<name<<endl;

cout<<"Id:"<<id<<endl;

}

};

int main(){

Teacher t1;

Teacher t2("Bibek", 78);

Teacher t3(t1);

t1.display();

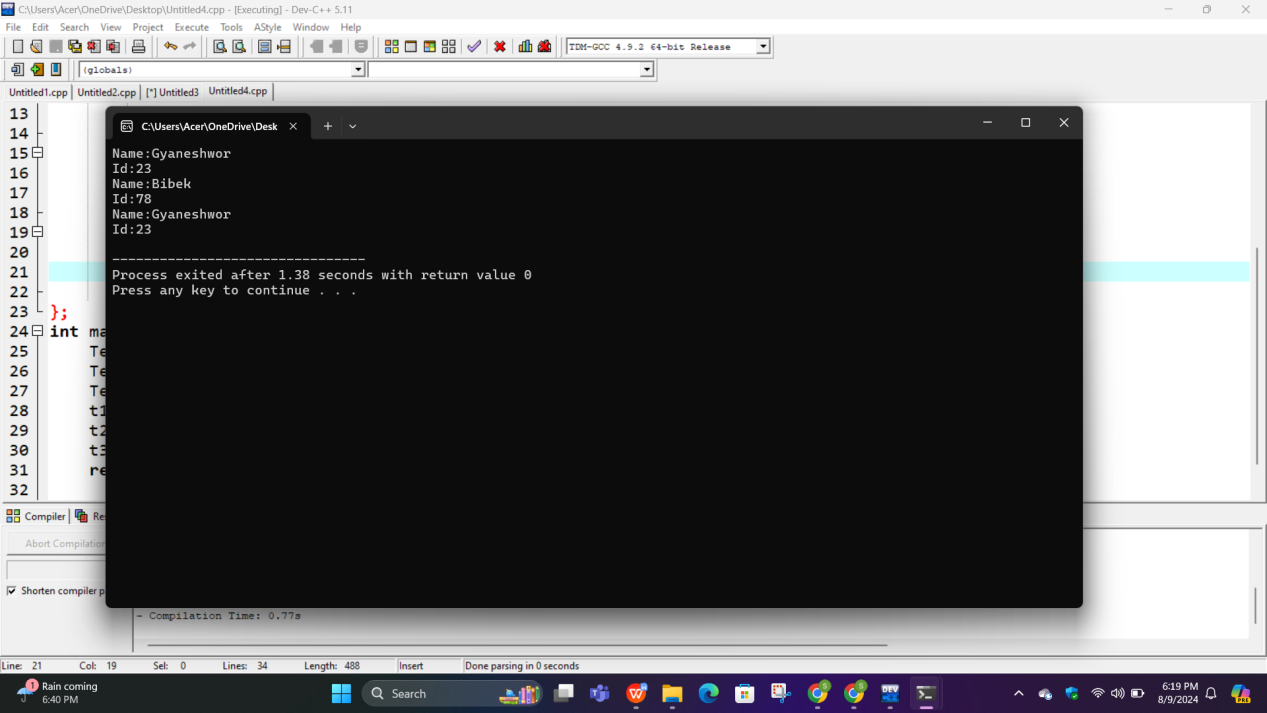
t2.display();

t3.display();

return 0;

}

**OUTPUT:**

****

**CONCLUSION:**

In this lab report, we explored the concept of function overloading in C++. Function overloading allows us to define multiple functions with the same name but different parameter lists, improving code organization and flexibility. function overloading within an object-oriented context, enhances the code organization and reusability.

**LAB 7:TO ILLUSTRATE THE CONCEPT OF UNARY AND BINARY OPERATOR IN C++.**

**OBJECTIVE:**

1.Understand the concept of operator overloading in C++.

2.Implement unary and binary operator overloading for a user-defined class.

4.Demonstrate how to use overloaded operators to perform custom operations on objects of the user-defined class.

**THEORY:**

In this lab session, we will explore the concept of operator overloading in C++. The primary focus will be on understanding both unary and binary operator overloading and gaining practical experience through illustrative examples.

**Operator Overloading Overview:** Operator overloading is a feature in C++ that allows us to redefine the behavior of built-in operators for user-defined classes. This means we can create custom behaviors for operators such as +, -, \*, /, ++, --, and more when they are used with objects of our own classes.

**Unary Operator Overloading:** Unary operators work on a single operand. Examples include increment (++), decrement (--), and negation (-). The purpose of unary operator overloading is to customize how these operators behave when applied to objects of user-defined classes. For instance, we can redefine the behavior of the increment operator (++) to increment a custom counter class by a specific value.

**Binary Operator Overloading:** Binary operators, on the other hand, operate on two operands. Examples include addition (+), subtraction (-), multiplication (\*), division (/), and equality (==). Binary operator overloading enables us to define the behavior of these operators when used with objects of our custom classes. For example, we can implement custom addition for complex numbers by overloading the + operator.

**PROGRAM 1:WAP TO OVERLOAD THE ++ OPERATOR.**

**//SOURCE CODE:**

#include<iostream>

using namespace std;

class Person{

private:

string name;

int age;

public:

Person(string n , int a){

name=n;

age=a;

}

void display(){

cout<<"name:"<<name<<endl;

cout<<"age:"<<age<<endl;

}

void operator++(){

age++;

}

};

int main(){

Person p("Samikshya" , 19);

p.display();

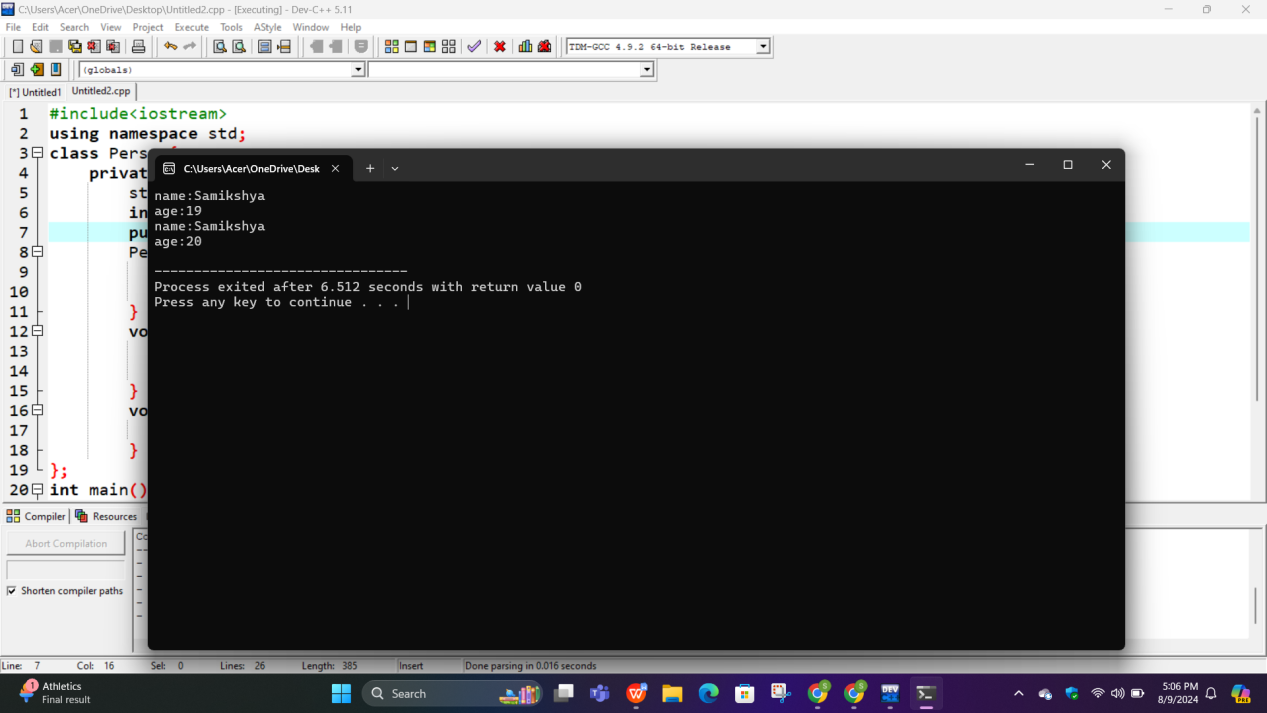
++p;

p.display();

return 0;

}

**OUTPUT:**

****

**PROGRAM 2:WAP TO OVERLOAD THE < OPERATOR TO COMPARE TWO PERSON BASED ON THEIR AGE.**

**//SOURCE CODE:**

#include <iostream>

using namespace std;

class Person {

private:

int age;

public:

Person(int a) : age(a) {}

Int operator<(Person& a){

return age < a.age;

}

};

int main() {

Person p1(95);

Person p2(30);

if (p1 < p2) {

cout << "Person 1 is younger than Person 2." <<endl;

} else {

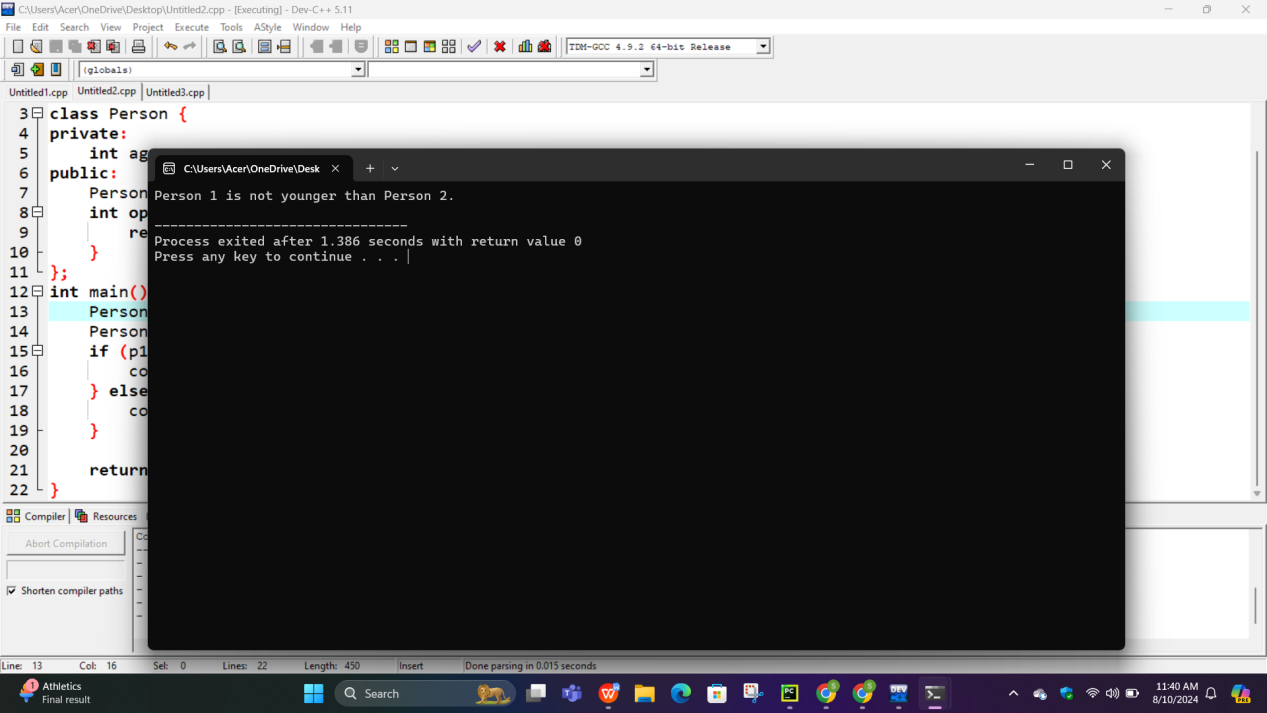
cout << "Person 1 is not younger than Person 2." <<endl;

}

return 0;

}

**OUTPUT:**

****

**CONCLUSION:**In this lab report, we explored the core concepts of unary and binary operator overloading in C++. Unary Operator Overloading was demonstrated with a age increment example, In summary, operator overloading is a valuable feature that enhances code expressiveness and flexibility in C++, allowing developers to create more user-friendly and intuitive custom classes.

**LAB 8:TO ILLUSTRATE THE CONCEPT OF INHERITANCE AND ITS TYPES IN C++.**

**OBJECTIVE:**

1.Understand the fundamental concept of inheritance in C++.

2. Explore the different types of inheritance supported in C++.

3. Implement simple inheritance relationships in C++ programs.

4.Gain practical experience in using inheritance to create derived classes from base classes.

**THEORY**: Inheritance is a fundamental concept in the world of C++ programming. It serves as a cornerstone of object-oriented programming, allowing for the creation of complex yet organized code structures. This lab aims to comprehensively explore the concept of inheritance in C++ and shed light on its various types through practical examples. Inheritance in C++: In C++, inheritance is the mechanism through which a new class, referred to as the derived class or subclass, can inherit the attributes and behaviors (i.e., data members and member functions) of an existing class known as the base class or superclass. This concept establishes an "is-a" relationship, where the derived class is essentially a specialized version of the base class, inheriting its characteristics while also having the ability to introduce new attributes or modify existing ones. The significance of inheritance in C++ is profound, as it paves the way for code reusability, modularity, and extensibility. It enables the encapsulation of common attributes and behaviors within a base class, allowing derived classes to acquire these traits through inheritance

**Types of Inheritance in C++**: C++ offers several types of inheritance, each designed for specific use cases. The primary types include:

**Single Inheritance**: Single inheritance is the simplest and most widely used form. It involves a derived class inheriting from a single base class. For instance, consider a base class called Shape, and derived classes like Circle, Rectangle, and Triangle, each inheriting from Shape. This allows them to share common attributes and methods while retaining their unique characteristics.

**Multiple Inheritance**: Multiple inheritance permits a derived class to inherit from multiple base classes. While powerful, it can introduce complexities, especially when base classes have common attributes or method signatures. For example, a class HybridVehicle might inherit from both ElectricVehicle and GasolineVehicle, gaining attributes from both parent classes.

**Multilevel Inheritance**: Multilevel inheritance involves the creation of a chain of inheritance, where a derived class is derived from another derived class. This results in a hierarchical structure. For example, a base class Animal may have a derived class Mammal, and further derived from Mammal, we can have Dog and Cat classes. Each level of inheritance adds more specialized attributes and methods.

**Hierarchical Inheritance**: In hierarchical inheritance, multiple derived classes inherit from a single base class. For example, a base class Employee might have derived classes like Manager, Engineer, and Salesperson. They share common attributes and methods defined in the base class but have their own unique properties.

**Hybrid Inheritance**: Hybrid inheritance is a combination of multiple types of inheritance. It can involve single, multiple, and multilevel inheritance within a single program. While it offers flexibility, it should be used judiciously to avoid code complexity. Each type of inheritance in C++ serves distinct purposes, allowing programmers to model relationships between classes according to the requirements of their applications.

**//SOURCE CODE:**

#include <iostream>

using namespace std;

// Base class for Single and Hierarchical Inheritance

class Person{

public:

void show() {

cout << "Base class function" << endl;

}

};

// Single Inheritance

class Good: public Person{

public:

void showSingleDerived() {

cout << "SingleDerived class function" << endl;

}

};

// Multiple Inheritance

class Rashmi{

public:

void showBase1() {

cout << "Base1 class function" << endl;

}

};

class Sofiya{

public:

void showBase2() {

cout << "Base2 class function" << endl;

}

};

class MultipleDerived : public Rashmi, public Sofiya{

public:

void showMultipleDerived() {

cout << "MultipleDerived class function" << endl;

}

};

// Multilevel Inheritance

class Level1 : public Person{

public:

void showLevel1() {

cout << "Level1 class function" << endl;

}

};

class Level2 : public Level1 {

public:

void showLevel2() {

cout << "Level2 class function" << endl;

}

};

// Hierarchical Inheritance

class HierarchicalDerived1 : public Person{

public:

void showHierarchical1() {

cout << "HierarchicalDerived1 class function" << endl;

}

};

class HierarchicalDerived2 : public Person{

public:

void showHierarchical2() {

cout << "HierarchicalDerived2 class function" << endl;

}

};

// Hybrid Inheritance

class HybridBase1 {

public:

void showHybridBase1() {

cout << "HybridBase1 class function" << endl;

}

};

class HybridBase2 {

public:

void showHybridBase2() {

cout << "HybridBase2 class function" << endl;

}

};

class HybridDerived : public HybridBase1, public HybridBase2, public Person{

public:

void showHybridDerived() {

cout << "HybridDerived class function" << endl;

}

};

int main() {

// Single Inheritance

Good sd;

sd.showSingleDerived();

// Multiple Inheritance

MultipleDerived md;

md.showBase1();

md.showBase2();

md.showMultipleDerived();

// Multilevel Inheritance

Level2 l2;

l2.showLevel1();

l2.showLevel2();

// Hierarchical Inheritance

HierarchicalDerived1 hd1;

hd1.showHierarchical1();

HierarchicalDerived2 hd2;

hd2.showHierarchical2();

// Hybrid Inheritance

HybridDerived hd;

hd.showHybridBase1();

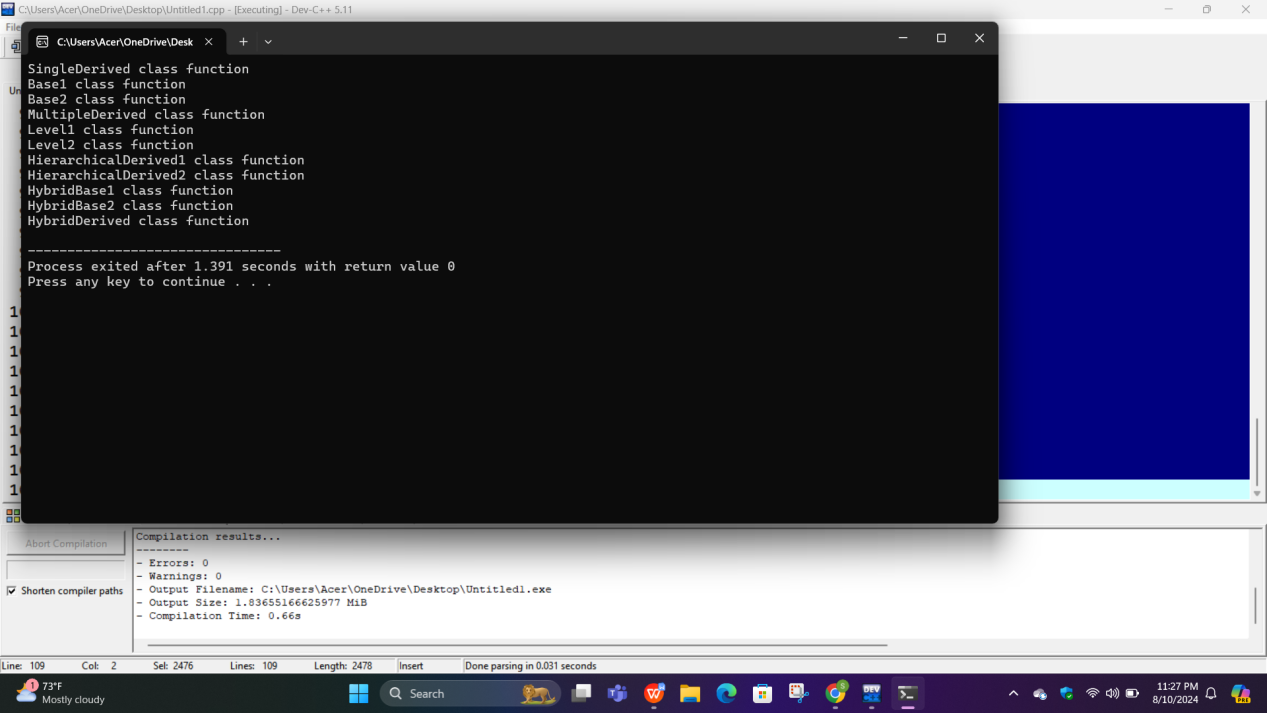
hd.showHybridBase2();

hd.showHybridDerived();

return 0;

}

**OUTPUT;**

****

**CONCLUSION:**

In this C++ program, we demonstrated the core concept of inheritance. Inheritance allows classes to inherit attributes and behaviors from other classes, promoting code reusability and structure. In summary, this program exemplifies the power of inheritance in C++ for organized, efficient, and extensible code. Understanding inheritance is vital for proficient object-oriented programming and building robust software systems.