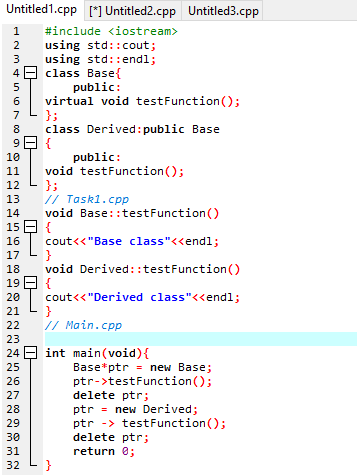


**Adan Bin Waheed**

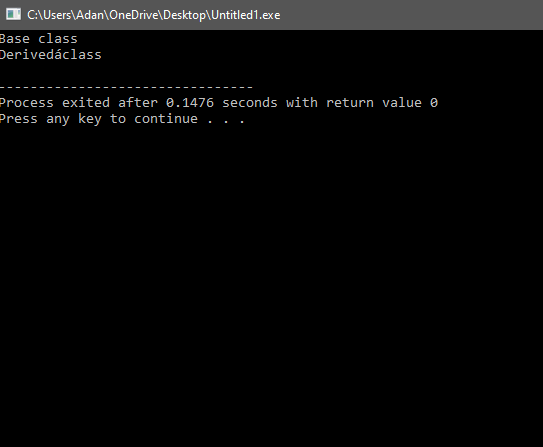
**45426**

**OOP – Lab Task - After Mids**

**Task 1.**

****

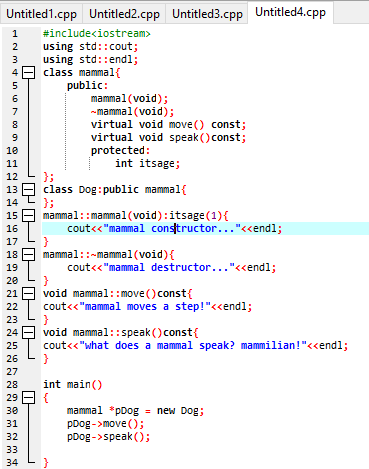
**Output**

****

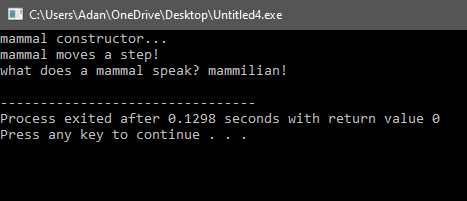
**Changing :**

**We Have Added a Virtual Function Because We have to Achieve polymorphism.**

**Task 2.**

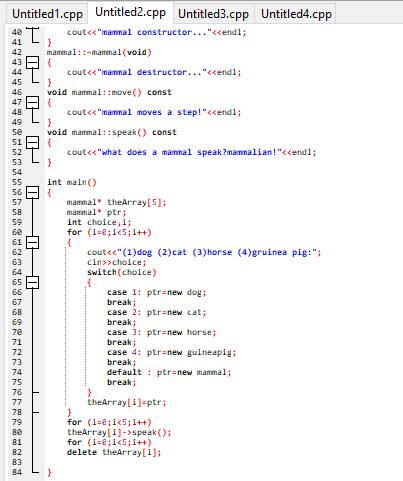
****

**Output**

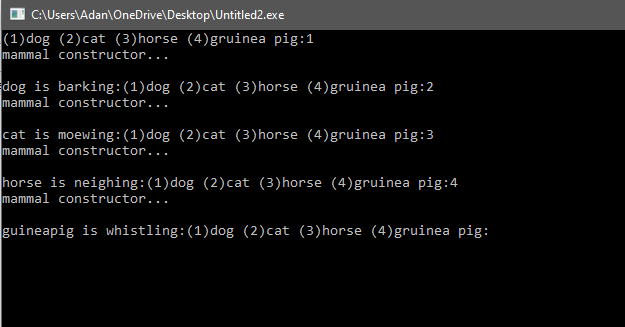
****

**Task 3.**

****

****

**Output**

****

**Q. Can a derived class make a public base function private?**

With private inheritance, public and protected member of the base class become private members of the derived class. That means the methods of the base class do not become the public interface of the derived object. However, they can be used inside the member functions of the derived class.

**Q. Why not make all class functions virtual?**

Virtual functions cannot be in-lined because inlining have to happen at runtime. This have performance impacts when you expect you functions benefits from inlining.

**Q. If a function (SomeFunc()) is virtual in a base class and is also overloaded, so as to take either an integer or two integers, and the derived class overrides the form taking one integer, what is called when a pointer to a derived object calls the two-integer form?**

When a pointer to a derived object calls the two-integer form of the overloaded virtual function that is defined in the base class, the version of the function that is called depends on how the derived class overrides the function. If the derived class overrides only the one-integer form of the function, and does not provide an implementation for the two-integer form, then the two-integer form in the base class will be called when invoked through a pointer to the derived class object.

**Here are some more questions:**

**Q. What is a v-table?**

To implement virtual functions, C++ uses a special form of late binding known as the virtual table. The virtual table is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.

**Q. What is a virtual destructor?**

Virtual Destructor in C++ is a member function that is used to free up the memory allocated used by the object of a child class or derived class when it is removed from the memory using the parent class pointer object.

**Q. How do you show the declaration of a virtual constructor?**

In C++, the constructor cannot be virtual, because when a constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet. So, the constructor should always be non-virtual.

But virtual destructor is possible.

**Q. How can you create a virtual copy constructor?**

**Virtual Constructor:-**  Not Possible because C++ is static type language and create constructor as a virtual so compiler won't be able to decide what type of object it and leave the whole process for run time because of virtual keyword. The compiler must be aware of the class type to create the object.

**Q. How do you invoke a base member function from a derived class in which you've overridden that function?**

To access the overridden function of the base class, we use the scope resolution operator :: . We can also access the overridden function by using a pointer of the base class to point to an object of the derived class and then calling the function from that pointer.

**Q. How do you invoke a base member function from a derived class in which you have not overridden that function?**

To modify the way a function defined in a base class works in the derived class, simply redefine the function in the derived class. Note that when you redefine a function in the derived class, the derived function does not inherit the access specifier of the function with the same name in the base class.

**Q. If a base class declares a function to be virtual, and a derived class does not use the term virtual when overriding that class, is it still virtual when inherited by a third-generation class?**

Yes, if a base class declares a function to be virtual, and a derived class overrides that function without using the virtual keyword, the function is still considered virtual. This means that when the function is called on an object of the derived class or any further derived class, the most-derived implementation of the function will be called, regardless of whether the virtual keyword was used in the overriding function or not.

**Q. What is the protected keyword used for?**

The protected keyword specifies access to class members in the member-list up to the next access specifier ( public or private ) or the end of the class definition.

**Some More Exercise**

**Q. Show the declaration of a virtual function that takes an integer parameter and returns void.**

virtual void myFunction(int myParam) = 0;

**Q. Show the declaration of a class Square, which derives from Rectangle, which in turn derives from Shape.**

class Shape {

public:

virtual double getArea() const = 0;

};

class Rectangle : public Shape {

public:

Rectangle(double width, double height) : width\_(width), height\_(height) {}

virtual double getArea() const override { return width\_ \* height\_; }

protected:

double width\_;

double height\_;

};

class Square : public Rectangle {

public: Square(double side) : Rectangle(side, side) {}

};

**Q. If, in Exercise 2, Shape takes no parameters, Rectangle takes two (length and width), but Square takes only one (length), show the constructor initialization for Square**

class Shape { public:Shape() {}

virtual double area() = 0;

};

class Rectangle : public Shape {

public:

Rectangle(double l, double w) : length(l), width(w) {}

virtual double area() { return length \* width; }

private:

double length;

double width;

};

class Square : public Rectangle {

public:

Square(double l) : Rectangle(l, l) {}

};

**Q. Write a virtual copy constructor for the class Square (in Exercise 3).**

class Shape {

public:

virtual Shape\* clone() const = 0;

virtual double area() const = 0;

virtual ~Shape() {}

};

class Rectangle : public Shape {

public:

Rectangle(double l, double w) : length(l), width(w) {}

virtual double area() const { return length \* width; }

virtual Rectangle\* clone() const { return new Rectangle(\*this); }

private:

double length;

double width;

};

class Square : public Rectangle {

public:

Square(double l) : Rectangle(l, l) {}

virtual Square\* clone() const { return new Square(\*this); }

};

**Q. BUG BUSTER : What is wrong with this code snippet?**

**Correction :**

void SomeFunction(Shape&);

Shape\* pRect = new Rectangle;

SomeFunction(\*pRect);

The issue with this code snippet is that the SomeFunction function takes its argument Shape by value instead of by reference or pointer. This means that when SomeFunction(\*pRect) is called, the Rectangle object pointed to by pRect is sliced, and only the Shape part of the object is passed to SomeFunction. As a result, any data or functionality specific to the Rectangle class will be lost, and SomeFunction will only be able to operate on the Shape part of the object.

**Q. BUG BUSTER : What is wrong with this code snippet?**

**Correction :**

class Shape {

public:

Shape() {}

virtual ~Shape() {}

Shape(const Shape &) {}

};

**There are a few issues with this code snippet:**

1. The constructor for Shape is declared with empty parentheses, but it doesn't need to be declared at all since the default constructor will be automatically generated by the compiler. If you want to define your own constructor, you should provide an implementation with a body and any necessary parameters.

2. The virtual destructor for Shape is declared correctly with the virtual keyword, but it shouldn't also be declared as a constructor with the same name. Constructors in C++ don't have a return type, so the line virtual Shape(const Shape &); is actually declaring a virtual function that takes a const Shape& parameter and returns a Shape object. This is not what you want for a constructor.

3. Constructors can't be virtual in C++, so the virtual keyword shouldn't be used on the constructor.