Institute of Geographical Information Systems

CS-212 - Object Oriented Programming LAB

Semester: Fall 2025	**Class: SCEE-IGIS - 2024**
Name: Ali Nawaz	**CMS ID : 00000526123**
Submitted to: Ma’am Alvina Anjum	**Due Date: Dec 03, 2025**

LAB 12: Polymorphism

Lab Task 1:

Build two classes, Mammal and Dog. Dog will inherit from Mammal. Below is the Mammal class code. Once you have the Mammal class built, build a second class Dog that will inherit publicly from Mammal.

```
#include <iostream>

using namespace std;

class Mammal

{

    public:

        Mammal(void);

        ~Mammal(void);

        virtual void Move() const;

        virtual void Speak() const;

    protected:

        int itsAge;

};

Mammal::Mammal(void):itsAge(1)

{
    cout << "Mammal constructor..." << endl;
}
Mammal::~Mammal(void)
{
    cout << "Mammal destructor..." << endl;
}
void Mammal::Move()
{
    cout << "Mammal moves a step!" << endl;
}

void Mammal::Speak()
{
    cout << "What does a mammal speak? Mammilian!" << endl;
}
```

Once you have completed class Mammal and Dog, build the following main program.

```
int main ()
{
  Mammal *pDog = new Dog;
  pDog->Move();
  pDog->Speak();

  //Dog *pDog2 = new Dog;

  //pDog2->Move();
  //pDog2->Speak();
  return 0;
}
```

What does it output, is that what you expected? Remove the keyword virtual from the class mammal and try it again. Now what happens? Next, put in another pointer to pDog2 in the main program, but this time make it a pointer to a Dog, not a mammal and create a new dog. Now what happens? What you should realize is that by making the method Speak virtual, we can have a little different behavior through dynamic (runtime) binding.

Answer:

- With `virtual`:

The output shows the **Dog** versions of `Move()` and `Speak()` being called. Yes, this is expected because virtual functions use runtime binding.

- Without `virtual`:

The program calls the **Mammal** versions of `Move()` and `Speak()` even though the object is a Dog. This happens because the pointer type is `Mammal*`, so no polymorphism occurs.

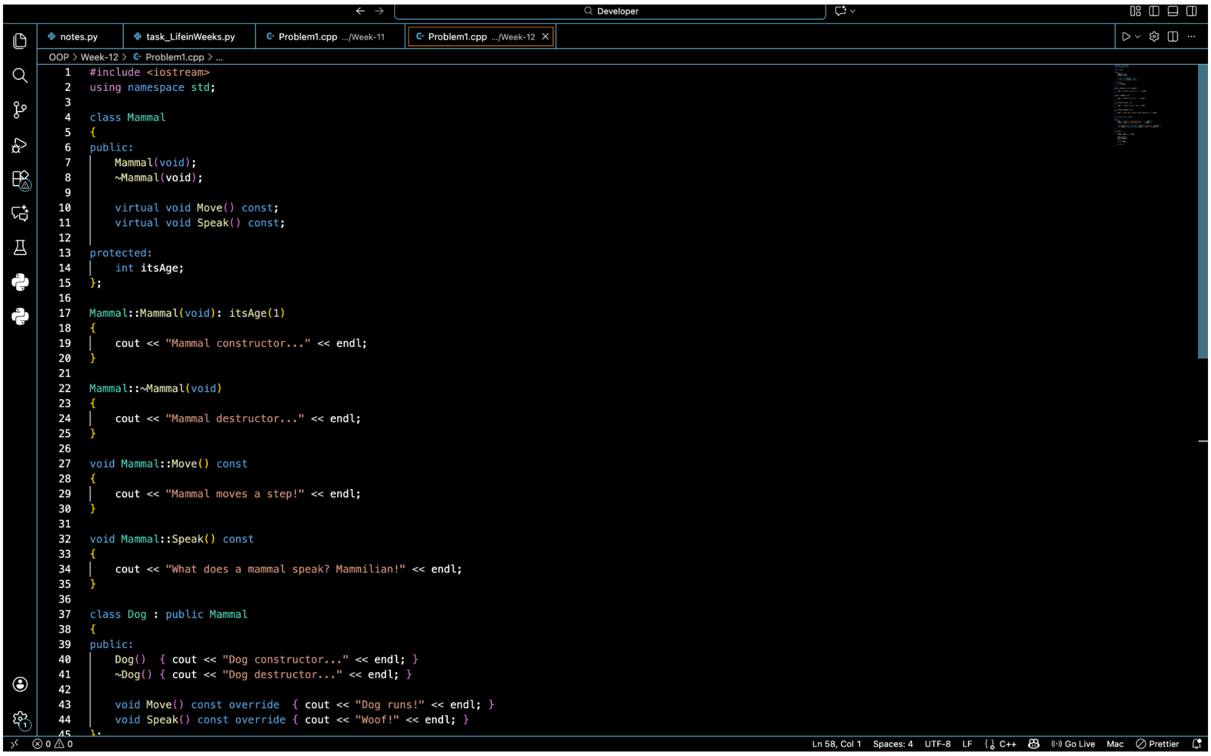
- Using `Dog* pDog2`:

When the pointer is actually a `Dog*`, the **Dog** versions of the functions run, even without virtual. This is because the pointer type matches the object type.

Conclusion:

Making the functions `virtual` allows C++ to choose the correct function at **runtime**, enabling true polymorphism.

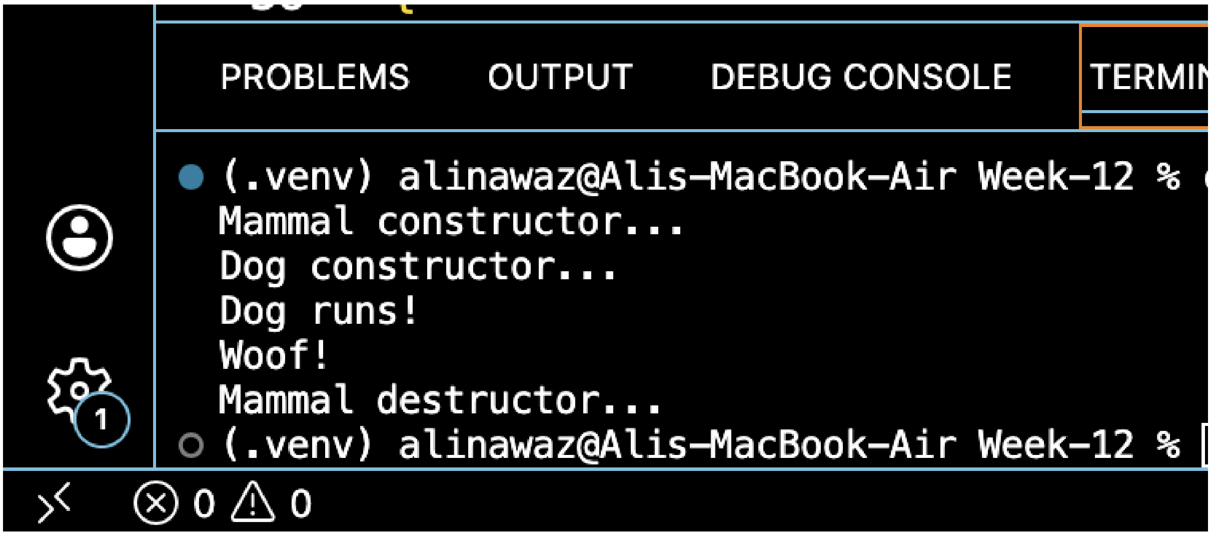
Screenshot:



Developer
notes.py
task_LifeinWeeks.py
Problem1.cpp .../Week-11
Problem1.cpp .../Week-12
OOP > Week-12 > Problem1.cpp > ...
Ln 58, Col 1   Spaces: 4   UTF-8   LF   {} C++   Go Live   Mac   Prettier
0 0

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

class Mammal
{
public:
    Mammal(void);
    ~Mammal(void);

    virtual void Move() const;
    virtual void Speak() const;

protected:
    int itsAge;
};

Mammal::Mammal(void): itsAge(1)
{
    cout << "Mammal constructor..." << endl;
}

Mammal::~Mammal(void)
{
    cout << "Mammal destructor..." << endl;
}

void Mammal::Move() const
{
    cout << "Mammal moves a step!" << endl;
}

void Mammal::Speak() const
{
    cout << "What does a mammal speak? Mammilian!" << endl;
}

class Dog : public Mammal
{
public:
    Dog()  { cout << "Dog constructor..." << endl; }
    ~Dog() { cout << "Dog destructor..." << endl; }

    void Move() const override  { cout << "Dog runs!" << endl; }
    void Speak() const override { cout << "Woof!" << endl; }
};
```

Output:



```
● (.venv) alinawaz@Alis-MacBook-Air Week-12 % 
  Mammal constructor...
  Dog constructor...
  Dog runs!
  Woof!
  Mammal destructor...
○ (.venv) alinawaz@Alis-MacBook-Air Week-12 % 
```

Lab Task 2:

Develop additional classes for Cat, Horse, and GuineaPig overriding the move and speak methods. (If you do not know guinea pigs go “weep weep”)

Next, test with the modified main:

```
int main ()
{
  Mammal* theArray[5];
  Mammal* ptr;
  int choice, i;
  for (i = 0; i<5; i++)
  {
    cout << "(1)dog (2)cat (3)horse (4)guinea pig: ";
    cin >> choice;
    switch (choice)
    {
      case 1: ptr = new Dog;
      break;
      case 2: ptr = new Cat;
      break;
      case 3: ptr = new Horse;
```

```
          break;
          case 4: ptr = new GuineaPig;
          break;
          default: ptr = new Mammal;
          break;
       }
       theArray[i] = ptr;
    }
    for (i=0;i<5;i++)
       theArray[i]->Speak();
// Always free dynamically allocated objects
     for (i=0;i<5;i++)
          delete theArray[i];
    return 0;
}
```

If the Dog object had a method, WagTail(), which is not in the Mammal, you could not use the pointer to Mammal to access that method (unless you cast it to be a pointer to Dog). Because WagTail() is not a virtual function, and because it is not in a Mammal object, you can't get there without either a Dog object or a Dog pointer to the Dog object!!!

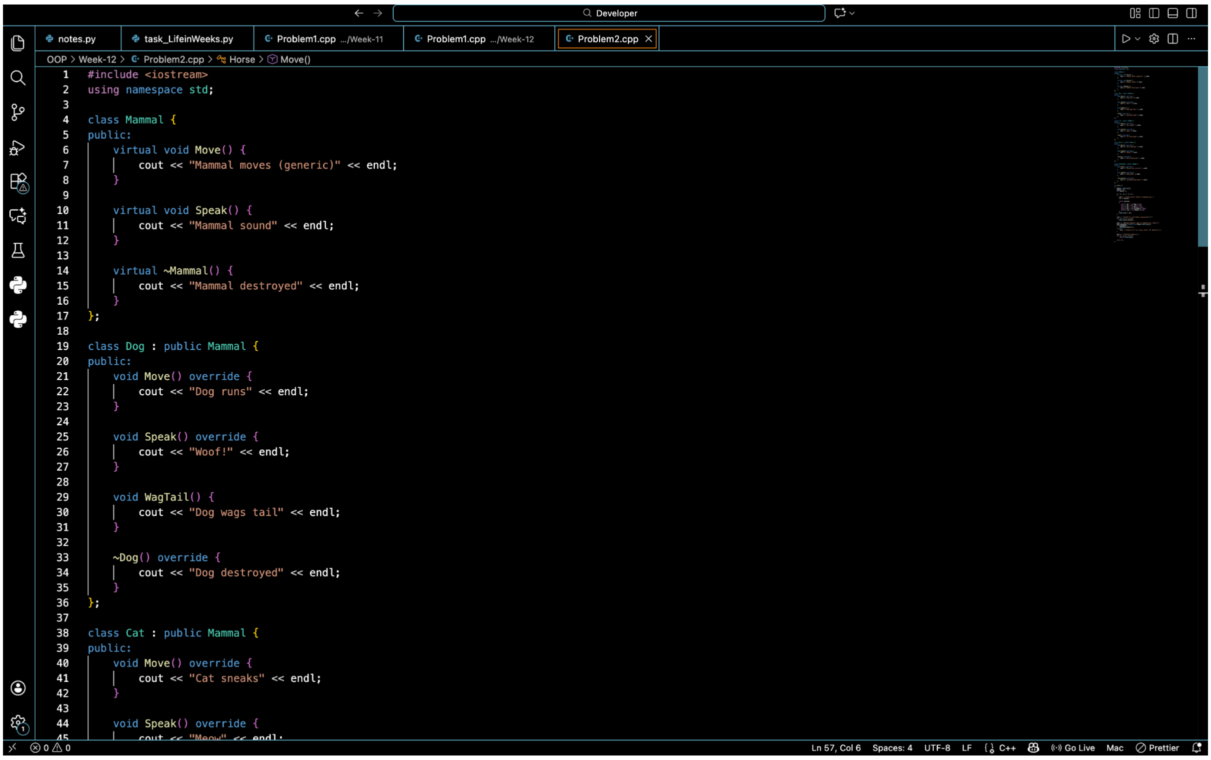
The virtual function magic (polymorphic behavior) operates only on pointers and references. Passing an object by value will not enable the virtual functions to be invoked.

Answer:

You can’t call WagTail() through a Mammal* because that function doesn’t exist in Mammal. To use it, you must cast the pointer to Dog* or use an actual Dog pointer.

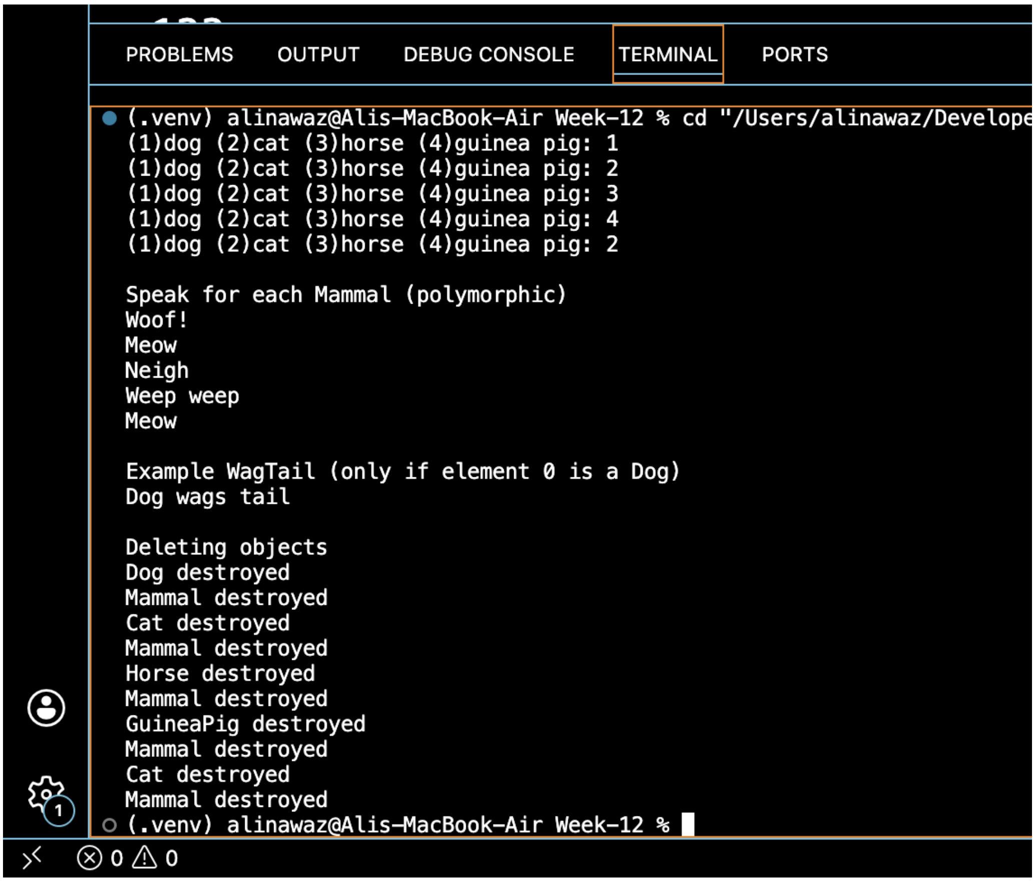
Virtual functions only work with pointers and references. If you pass an object by value, polymorphism is lost because the object gets copied and behaves like the base class.

Screenshot:



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

class Mammal {
public:
    virtual void Move() {
        cout << "Mammal moves (generic)" << endl;
    }

    virtual void Speak() {
        cout << "Mammal sound" << endl;
    }

    virtual ~Mammal() {
        cout << "Mammal destroyed" << endl;
    }
};

class Dog : public Mammal {
public:
    void Move() override {
        cout << "Dog runs" << endl;
    }

    void Speak() override {
        cout << "Woof!" << endl;
    }

    void WagTail() {
        cout << "Dog wags tail" << endl;
    }

    ~Dog() override {
        cout << "Dog destroyed" << endl;
    }
};

class Cat : public Mammal {
public:
    void Move() override {
        cout << "Cat sneaks" << endl;
    }

    void Speak() override {
        cout << "Meow" << endl;
```

Output:



```
(.venv) alinawaz@Alis-MacBook-Air Week-12 % cd "/Users/alinawaz/Develope
(1)dog (2)cat (3)horse (4)guinea pig: 1
(1)dog (2)cat (3)horse (4)guinea pig: 2
(1)dog (2)cat (3)horse (4)guinea pig: 3
(1)dog (2)cat (3)horse (4)guinea pig: 4
(1)dog (2)cat (3)horse (4)guinea pig: 2

Speak for each Mammal (polymorphic)
Woof!
Meow
Neigh
Weep weep
Meow

Example WagTail (only if element 0 is a Dog)
Dog wags tail

Deleting objects
Dog destroyed
Mammal destroyed
Cat destroyed
Mammal destroyed
Horse destroyed
Mammal destroyed
GuineaPig destroyed
Mammal destroyed
Cat destroyed
Mammal destroyed
(.venv) alinawaz@Alis-MacBook-Air Week-12 %
```

1. Are inherited members passed to later generations?

Ans. Yes. If Dog → Mammal → Animal, then Dog gets everything from Mammal and Animal, unless something is private or restricted.

2. If Mammal overrides a function from Animal, which one does Dog get?

Ans. Dog gets the overridden version in Mammal, unless Dog overrides it again.

3. Can a derived class make a public base function private?

Ans. Yes. A derived class can change the access level of inherited functions.
But this only affects access through Dog, not through Animal/Mammal pointers.

4. Why not make all functions virtual?

Ans. Because virtual functions have a small performance cost, use extra memory (vtable), and sometimes you don't need polymorphism.
So we only make functions virtual when they should be overridden.

5. If only the 1-integer version is overridden, what happens to the 2-integer version?

Ans. The 2-integer version from the base class is called, because the derived class did not override that version.

Lab Task 3: Virtual Destructor Experiment

Objective:

Understand why destructors must be virtual when deleting derived class objects using base class pointers.

Instructions:

1. Create a base class Shape with:

o Virtual Draw()

o A non-virtual destructor that prints "Shape destroyed"

2. Create a derived class Circle with:

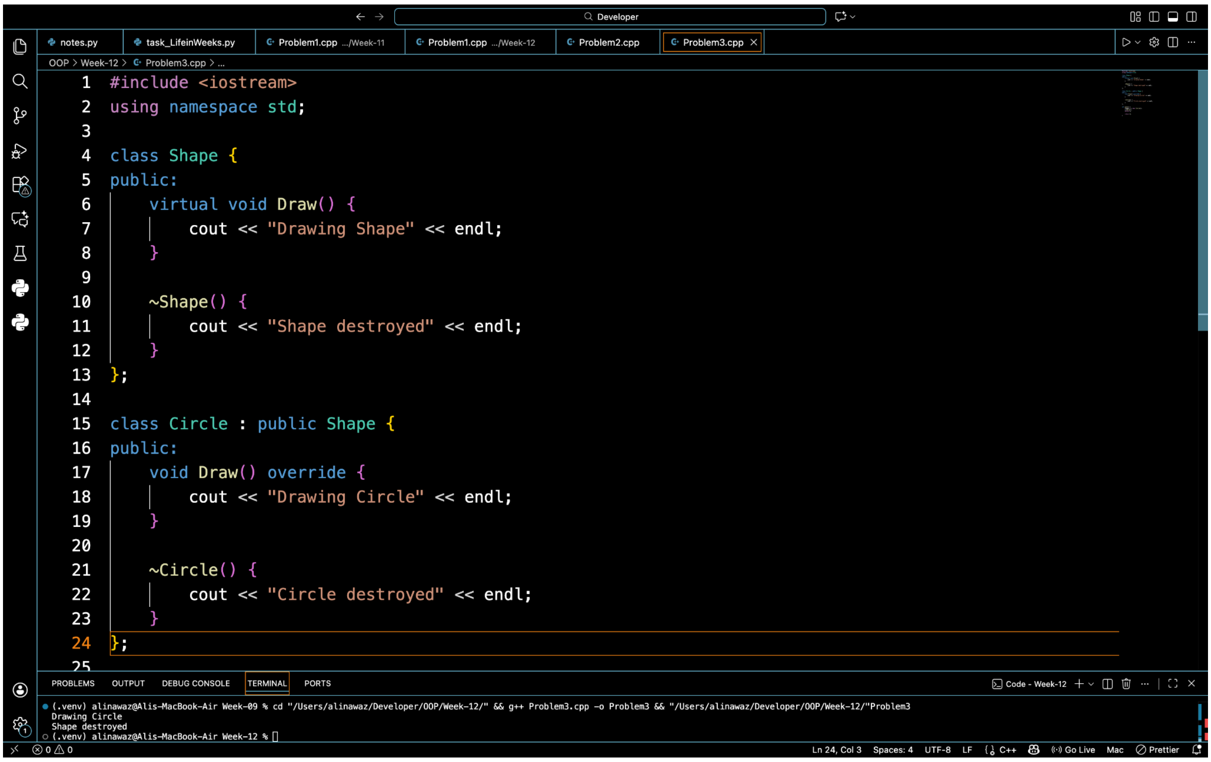
o Overridden Draw()

o Destructor printing "Circle destroyed"

3. Write the following main code:

```
Shape* s = new Circle();

s->Draw();

delete s;
```

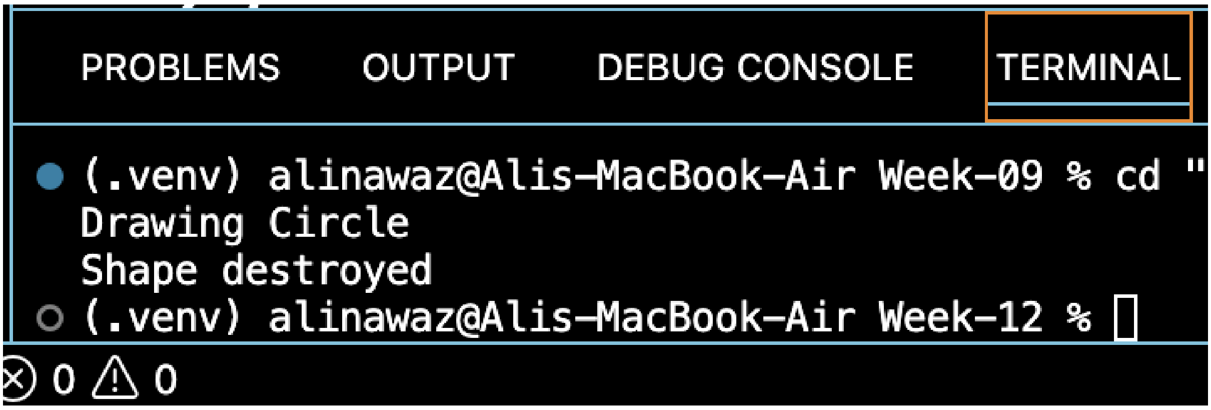
Screenshot:



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

class Shape {
public:
    virtual void Draw() {
        cout << "Drawing Shape" << endl;
    }

    ~Shape() {
        cout << "Shape destroyed" << endl;
    }
};

class Circle : public Shape {
public:
    void Draw() override {
        cout << "Drawing Circle" << endl;
    }

    ~Circle() {
        cout << "Circle destroyed" << endl;
    }
};

```

```
● (.venv) alinawaz@Alis-MacBook-Air Week-09 % cd "/Users/alinawaz/Developer/OOP/Week-12/" && g++ Problem3.cpp -o Problem3 && "/Users/alinawaz/Developer/OOP/Week-12/"Problem3
  Drawing Circle
  Shape destroyed
○ (.venv) alinawaz@Alis-MacBook-Air Week-12 % ▯
```

Output:



```
● (.venv) alinawaz@Alis-MacBook-Air Week-09 % cd "
  Drawing Circle
  Shape destroyed
○ (.venv) alinawaz@Alis-MacBook-Air Week-12 % ▯
```

1. Which destructors run?

Ans. Only the Shape destructor runs. The Circle destructor does not run because the base class destructor is non-virtual.

2. Why is the Circle destructor not called?

Ans. Because the destructor in Shape is not virtual.
When we write:
Shape* s = new Circle();
delete s;
The delete operation uses the static type of the pointer (Shape*) and calls only Shape's destructor. And since it is non-virtual, C++ does not look for the derived destructor, so the Circle destructor is skipped.

3. Make the destructor in Shape virtual. Now what happens?

Ans. After changing the destructor to:
virtual ~Shape()
and running the same code:
delete s;

Both destructors run, first Circle, then Shape.
This ensures the object is fully destroyed in the correct order.

4. Why is a virtual destructor essential in polymorphic base classes?

Ans. A virtual destructor ensures that when a derived object is deleted through a base-class pointer:

- The derived destructor runs first,
- Then the base destructor runs,
- All resources owned by the derived class are properly released,
- No memory leaks or incomplete cleanup occur.

In short:
If a class is used polymorphically, its destructor must be virtual to allow safe deletion through base pointers.