Q5. Written Questions Answers

(A) virtual & override

The `virtual` keyword in the base class Trade allows for dynamic polymorphism. When a member function is declared as virtual in the base class, the actual function called is determined at runtime based on the type of the object pointed to by the base class pointer.

For example: CPP code:-

```
Trade* t = new Call(1.0, 6.0);
t->payoff(10); // Calls Call's payoff function
```

If `payoff` wasn't virtual, it would call Trade's payoff function instead of Call's.

Declaring a function as pure virtual (with `= 0`) makes the class abstract, meaning it cannot be instantiated. Derived classes must implement this function.

The `override` specifier in derived classes ensures that the function is actually overriding a virtual function from the base class. It doesn't change the generated code but helps catch errors at compile time.

(B) Polymorphism

Static polymorphism is resolved at compile time (using function overloading), while dynamic polymorphism is resolved at runtime (using virtual functions).

In the Trade example, dynamic polymorphism allows us to store different types of trades (Forward, Call, Put) in a container of Trade pointers and call their respective payoff functions without knowing their exact type at compile time.

(C) Compiler issues and optimizations

When we uncomment the destructor in Widget, the compiler no longer generates the move constructor and move assignment operator. This is because defining a destructor tells the compiler that we want to handle resource management ourselves, so it doesn't generate the special member functions.

Line 22 and 24 still work because the copy constructor is still generated by the compiler. When we define our own destructor, the compiler will still generate the copy constructor unless we explicitly delete it.

(D) Compiler issues and optimizations 2

In the first case, when returning without `std::move()`, the compiler performs copy elision, so no copy or move constructor is called.

When we use `std::move()`, we explicitly tell the compiler to treat the object as an rvalue, allowing the move constructor to be used.

When we comment out the move constructor, the compiler falls back to using the copy constructor.

(E) Misc.

(i) Virtual inheritance is needed when you want to ensure that only one instance of a base class exists in the inheritance hierarchy, even if the class is inherited multiple times through different paths.

(ii) The singleton pattern ensures that a class has only one instance and provides a global point of access to it. In C++, this can be implemented by making the constructor private and providing a static member function that returns a reference to the single instance.

Q6. AI example for polymorphism

```
#include <iostream>
#include <vector>

// Base class
class Shape {
public:
    virtual ~Shape() = default;
    virtual double area() const = 0;
};

// Derived class 1
class Circle : public Shape {
public:
    Circle(double r) : radius(r) {}
    double area() const override {
        return 3.14159 * radius * radius;
    }
private:
    double radius;
};

// Derived class 2
class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double area() const override {
        return width * height;
    }
private:
    double width;
    double height;
};

// Function that uses polymorphism
void print_areas(const std::vector<Shape*>& shapes) {
    for (const auto& shape : shapes) {
        std::cout << "Area: " << shape->area() << std::endl;
    }
}

int main() {
    std::vector<Shape*> shapes;
    shapes.push_back(new Circle(5.0));
    shapes.push_back(new Rectangle(4.0, 6.0));

    print_areas(shapes);

    // Clean up
    for (auto& shape : shapes) {
        delete shape;
    }

    return 0;
}
```

This example demonstrates polymorphism by having a base class `Shape` with a pure virtual function `area()`. Derived classes `Circle` and `Rectangle` implement this function according to their specific geometry. The function `print_areas` takes a vector of `Shape*` and calls `area()` on each pointer, which dynamically dispatches to the correct implementation based on the actual object type.

The base class `Shape` is an abstract base class because it has a pure virtual function. The inheritance is public, which means the derived classes inherit the interface and implementation access correctly. The `override` keyword ensures that we're actually overriding a virtual function from the base class.

Polymorphism is useful here because it allows us to treat different shapes uniformly through a common interface. This makes the code more flexible and extensible - we can add new shapes without modifying the `print_areas` function.