Name: ______________________________

2143 OOP Exam 2
Spring 2025

1. A class variable is declared as `static` and only one copy exists for the entire class, whereas an instance variable is unique to each instance of the class. Which of the following statements is correct?
   - **A. Changing the class variable in one instance affects the variable for all instances.**
   - B. Changing the instance variable in one instance affects the variable for all instances.
   - C. Both class and instance variables are shared among all instances.
   - D. Neither class variables nor instance variables are shared.

2. Which of the following statements best describes pure polymorphism in C++?
   - A. It is achieved by function overloading within a single class.
   - **B. It is achieved by defining a base class with one or more pure virtual functions, forcing derived classes to provide their own implementations.**
   - C. It is achieved by using non-virtual functions that are inherited by derived classes.
   - D. It is achieved by operator overloading in derived classes.

3. Fill in the blanks: An object is a(n) ________ of a class that resides in ________ and has ________.
   - **A. instance, memory, state**
   - B. instance, disk, state
   - C. class, memory, state
   - D. instance, heap, properties and methods

4. Which keywords can be used to control access to class members?
   - A. class, struct, union
   - **B. public, protected, private**
   - C. static, const, volatile
   - D. friend, inline, virtual

5. What type of data member can be shared by all instances of its class?
   - A. Instance data member
   - B. Reference data member
   - C. Constant data member
   - **D. Static data member**

6. What is the difference between private and protected access?
   - A. Private members can be accessed by derived classes, whereas protected members cannot.
   - **B. Protected members can be accessed by derived classes, whereas private members cannot.**
   - C. Both private and protected members are accessible to derived classes.
   - D. There is no difference between private and protected access.

7. What does the `friend` keyword in C++ allow us to do?
   - A. It restricts the access of class members to within the class.
   - **B. It allows a function or another class to access private and protected members.**
   - C. It automatically makes all class members public.
   - D. It allows a class to inherit from another class.

8. What is a potential drawback of using the `friend` keyword?
   - A. It always leads to memory leaks.
   - B. It makes code run significantly slower.
   - C. It prevents the use of inheritance.
   - **D. It can break encapsulation and increase coupling between classes.**

9. (**Refer to Snippet 1**) What is the purpose of making ‘secretData‘ private in the ‘Base‘ class?
    A. To allow ‘Derived‘ class to access and modify ‘secretData‘.
    B. To make ‘secretData‘ accessible to all classes in the program.
    C. To enforce strict encapsulation and ensure that ‘secretData‘ is only managed internally by the ‘Base‘ class.
    D. To prevent the ‘Base‘ class from accessing its own data members.

10. What is the output of the program in textbfSnippet 2?
    A. Playing generic animal sound...
    B. Dogs bark...
    C. Compilation error
    D. Runtime error

11. What concept of oop is demonstrated by textbfSnippet 2?
    A. Encapsulation
    B. Inheritance
    C. Polymorphism
    D. Abstraction

12. What is the one thing that is necessary for run-time polymorphism to happen?
    A. Inheritance only.
    B. Function overloading.
    C. Virtual functions.
    D. Operator overloading.

13. Give an example of compile time polymorphism.
    A. Function overloading.
    B. Virtual function overriding.
    C. Runtime type identification.
    D. Dynamic binding.

14. How can we make a class abstract in C++?
    A. By declaring all member functions as virtual.
    B. By declaring at least one member function as pure virtual.
    C. By using the keyword `abstract` in the class definition.
    D. By inheriting from an `Abstract` class.

15. What is the purpose of an abstract class in C++?
    A. To allow direct instantiation of objects.
    B. To provide an interface and force derived classes to implement specific functions.
    C. To enhance memory management.
    D. To speed up program execution.

16. How many objects can be created from an abstract class?
    A. One.
    B. Many.
    C. Zero.
    D. Two.

17. What is the difference between an abstract class and an interface in C++?
    A. An abstract class can have both pure virtual and implemented functions, whereas an interface typically contains only pure virtual functions and no data members.
    B. An interface can have member data, while an abstract class cannot.
    C. There is no difference; they are implemented in exactly the same way.
    D. An abstract class is used only for run-time polymorphism, while an interface is used only for compile-time polymorphism.

18. What does it mean: “The use of an object of one class in the definition of another class”?

A. It signifies an inheritance relationship.
B. It signifies a loose association.
C. It signifies composition, meaning one class contains an object of another class.
D. It signifies encapsulation of data.

19. Is there ever a time you would use private in a base class and prevent access to a derived class? Consider **Snippet 4**:
What is the purpose of making ‘secretData‘ private in the ‘Base‘ class?
A. To allow ‘Derived‘ class to access and modify ‘secretData‘.
B. To make ‘secretData‘ accessible to all classes in the program.
C. To enforce strict encapsulation and ensure that ‘secretData‘ is only managed internally by the ‘Base‘ class.
D. To prevent the ‘Base‘ class from accessing its own data members.

20. Consider the code in **Snippet 2**:
What is the output of the program?
A. Playing generic animal sound...
B. Dogs bark...
C. Compilation error
D. Runtime error

21. What concept of object-oriented programming is demonstrated by **Snippet 2** as well?
A. Encapsulation
B. Inheritance
C. Polymorphism
D. Abstraction

22. What does it mean: “The use of an object of one class in the definition of another class”?
A. It signifies an inheritance relationship.
B. It signifies a loose association.
C. It signifies composition.
D. It signifies encapsulation of data.

23. Which of the classes demonstrated inheritance in **Snippet 3**?
A. Neither
B. Account
C. Car
D. Both

24. Which of the classes demonstrated encapsulation in **Snippet 3**? as well?
A. Neither
B. Account
C. Car
D. Both

25. Consider the code in **Snippet 5** for the next two questions:
Is the ‘print‘ method in the ‘Wizard‘ class overloading or overriding the ‘print‘ method from the ‘Character‘ class?
A. Overloading
B. Overriding
C. Neither
D. Both

26. Is the ‘print‘ method in the ‘Character‘ class itself an example of overloading, overriding, or neither?
A. Overloading
B. Overriding
C. Neither
D. Both

27. A program for a card game wants to keep track of certain game scores and make them accessible to every team instance. Which concept best suits this need?
    A. Static data member : Because static members are shared across all instances.
    B. Instance data member : These are unique to each instance.
    C. Global variable : Though accessible everywhere, globals break encapsulation.
    D. Dynamic allocation : This does not inherently share data among instances.

28. In a multimedia application, the base class `Media` is designed so that derived classes like `Audio`, `Video`, and `Image` must provide their own implementation of the `play()` method. Which concept best enforces this requirement?
    A. Function overloading : This allows multiple functions with the same name but does not enforce implementation.
    B. Pure virtual function : Declaring `play()` as pure virtual in `Media` forces derived classes to implement it.
    C. Static function : This is unrelated to enforcing derived class implementations.
    D. Friend function : This merely grants access, not implementation enforcement.

29. A graphics application has a base class `Shape` with a function `draw()`. Derived classes such as `Circle`, `Square`, and `Triangle` each provide their own version of `draw()`. Which concept does this exemplify?
    A. Function overloading : This involves functions with the same name but different parameters.
    B. Overriding : Derived classes are replacing the base class version of `draw()` with their own implementations.
    C. Hiding : This refers to non-virtual functions that are re-declared in a derived class.
    D. Constructor initialization : This is unrelated to method behavior in inheritance.

30. Which of the following can be overloaded in C++?
    A. Constructors
    B. Operators
    C. Methods
    D. All of the above
    E. None of the above

31. Which of the following items can we override in C++?
    A. Constructors
    B. Operators
    C. Methods
    D. All of the above
    E. None of the above

32. Which protection specifier allows class members to be visible within the same class, in derived classes, and also outside the class (for non-friend code)?
    A. public
    B. protected
    C. private
    D. none of the above

33. Which of the following protection specifiers restricts access so that class members are visible only within the class itself, and not in derived classes or outside the class?
    A. public
    B. protected
    C. private
    D. both protected and private

34. A class designer wants to expose certain members only to derived classes while keeping them hidden from code outside the class. Which protection specifier best fits this requirement?
    A. public
    B. protected
    C. private
    D. friend

35. Which of the following best describes the typical use case for members declared as private?
    A. To form the public API for other parts of the program.
    B. To allow derived classes to access internal mechanisms.
    C. To hold strictly internal data that is hidden from both derived classes and non-friend external code.
    D. To enable global access while maintaining encapsulation.

36. What are the visibility characteristics of members declared as protected in a class?
    A. They are visible within the class and accessible to non-friend external code.
    B. They are visible within the class and in derived classes, but not accessible outside the class for non-friend code.
    C. They are only visible within the class.
    D. They are visible everywhere, similar to public members.

37. Are there any classes in Snippet 1 or 2 that wouldn't be considered a concrete class?
    A. None of them are concrete classes
    B. They are all concrete classes
    C. class Dogs and class Derived
    D. class Base and class Animals

38. Which of the following statements best describes **aggregation** in object-oriented design?
    A. Aggregation is a relationship where the container fully controls the lifetime of the contained object.
    B. Aggregation is the same as composition.
    C. Aggregation is a weak association where a class holds a reference to another class, and the contained object can exist independently.
    D. Aggregation is a mechanism to inherit functionality from multiple classes.

39. Consider a scenario where a `Library` class maintains a collection of `Book` objects. The `Book` objects can exist independently of the `Library` (e.g., they can be transferred to another library). Which type of relationship is this an example of?
    A. Composition, because the `Library` creates and owns the `Book` objects.
    B. Aggregation, because the `Book` objects can exist independently of the `Library`.
    C. Inheritance, because `Book` is a type of `Library` item.
    D. Encapsulation, because the `Library` hides the `Book` details.

40. How does **composition** differ from **aggregation** in the context of object relationships?
    A. In composition, objects can be shared among multiple owners.
    B. In composition, the contained object's lifetime is independent of the container.
    C. In composition, the container controls the lifetime of the contained object, so if the container is destroyed, so are its parts.
    D. Composition allows for a weak association, while aggregation creates a strong dependency.

Snippet 1

```
// Snippet 1
////////////////////////////////////////////////////////////////////////////////

class Base {
private:
    int secretData; // Only accessible within Base
public:
    Base(int data) : secretData(data) {}
};

class Derived : public Base {
public:
    Derived(int data) : Base(data) {}
    void showSecret() {
        // Cannot access secretData here because it is private in Base
    }
};
```

Snippet 2

```
// Snippet 2
class Animals {
public:
    virtual void sound() {
        cout << "Playing generic animal sound..." << endl;
    }
};

class Dogs : public Animals {
public:
    void sound() {
        cout << "Dogs bark..." << endl;
    }
};
int main() {
    Animals *a;
    Dogs d;
    a = &d;
    a->sound();
    return 0;
}
```

Snippet 3

```

class Vehicle {
public:
    void start() {
        cout << "Vehicle started" << endl;
    }
};

class Car : public Vehicle {
public:
    void drive() {
        cout << "Car is driving" << endl;
    }
};

class Account {
private:
    double balance;
public:
    Account(double initial) : balance(initial) {}
    double getBalance() const {
        return balance;
    }
    void deposit(double amount) {
        if (amount > 0)
            balance += amount;
    }
};

int main() {
    Account acc(100.0);
    acc.deposit(50.0);
    cout << acc.getBalance() << endl;

    Car myCar;
    myCar.start();
    myCar.drive();
    return 0;
}
```

Snippet 4

```
class Base {
private:
    int secretData;
public:
    Base(int data) : secretData(data) {}
};

class Derived : public Base {
public:
    Derived(int data) : Base(data) {}
    void showSecret() {

    }
};
```

Snippet 5

```
    class Character {
    protected:
        string name;
    public:
        void print() {
            cout << name << endl;
        }
    };

    class Wizard : public Character {
    public:
        void print() {
            cout << name << " is a Wizard!" << endl;
        }
    };


    class Animal {
    public:
```

```
    virtual void makeSound() const {
        std::cout << "Generic animal sound\n";
    }
};

class Dog : public Animal {
public:
    void makeSound() const override {
        std::cout << "Woof!\n";
    }
};

class Person {
private:
    Animal* pet;
public:
    Person(Animal* p) : pet(p) { }
    void playWithPet() const {
        std::cout << "Playing with pet: ";
        pet->makeSound();
    }
};

class Child{
private:
    Person* parent;
public:
    Child(Person* p) : parent(p) { }
    void playWithParent() const {
        std::cout << "Playing with parent: ";
        parent->playWithPet();
    }
}
```