**Chapter 20: Object-Oriented Programming: Polymorphism**

**Section 20.1 Introduction**

20.1 Q1: Polymorphism is implemented via:

1. Member functions.
2. virtual functions and dynamic binding.
3. inline functions.
4. Non-virtual functions.

**ANS: b. virtual functions and dynamic binding.**

**Section 20.2 Introduction to Polymorphism: Polymorphic Video Game**

20.2 Q1: Which of the following statements about polymorphism is *false*?

1. With polymorphism, you can direct a variety of objects to behave in manners appropriate to those objects without even knowing their types.
2. With polymorphism, new types of objects that can respond to existing messages can easily be incorporated into a system without modifying the base system.
3. Polymorphism enables you to deal in specifics and let the execution-time environment concern itself with the generalities.
4. To get polymorphic behavior among existing objects, those objects must be instantiated from classes in the same inheritance hierarchy.

**ANS: c. Polymorphism enables you to deal in specifics and let the execution-time environment concern itself with the generalities. Actually, the reverse is true.**

**Section 20.3 Relationships Among Objects in an Inheritance Hierarchy**

20.2 Q1: Which of the following would not be a member function that derived classes Fish, Frog and Bird should inherit from base class Animal and then provide their own definitions for, so that the function call can be performed polymorphically?

1. eat
2. sleep
3. move
4. flapWings

**ANS: d. flapWings**

**Section 20.3.1 Invoking Base-Class Functions from Derived-Class Objects**

20.3.1 Q1: Employee is a base class and HourlyWorker is a derived class, with a redefined non-virtual print function. Given the following statements, will the output of the two print function calls be identical?

HourlyWorker h;

Employee \*ePtr = &h;

ePtr->print();

ePtr->Employee::print();

1. Yes.
2. Yes, if print is a static function.
3. No.
4. It would depend on the implementation of the print function.

**ANS: a. Yes.**

**Section 20.3.2 Aiming Derived-Class Pointers at Base-Class Objects**

20.3.2 Q1: Which of the following assignments would be a compilation error?

1. Assigning the address of a base-class object to a base-class pointer.
2. Assigning the address of a base-class object to a derived-class pointer.
3. Assigning the address of a derived-class object to a base-class pointer.
4. Assigning the address of a derived-class object to a derived-class pointer.

**ANS: b. Assigning the address of a base-class object to a derived-class pointer.**

**Section 20.3.3 Derived-Class Member-Function Calls via Base-Class Pointers**

20.3.3 Q1: Downcasting enables:

1. A derived-class object to be treated as a base-class object.
2. A base-class object to be treated as a derived-class object.
3. Making a base-class pointer into a derived-class pointer.
4. Making a derived-class pointer into a base -class pointer.

**ANS: c. Making a base-class pointer into a derived-class pointer.**

**Section 20.3.4 Virtual Functions and Virtual Destructors**

20.3.4 Q1: If objects of all the classes derived from the same base class all need to draw themselves, the draw function would most likely be declared:

1. private
2. virtual
3. protected
4. friend

**ANS: b. virtual**

20.3.4 Q2: virtual functions must:

1. Be overridden in every derived class.
2. Be declared virtualin every derived class.
3. Be declared virtual in the base class.
4. Have the same implementation in *every* derived class.

**ANS: c. Be declared virtual in the base class.**

20.3.4 Q3: Which of the following statements about virtual functions is *false*?

1. They allow the program to select the correct implementation at execution time.
2. They can use either static or dynamic binding, depending on the handles on which the functions are called.
3. They do not remain virtual down the inheritance hierarchy.
4. They can be called using the dot operator.

**ANS: c. They do not remain virtual down the inheritance hierarchy.**

20.3.4 Q4[C++11]: Which of the following statements is *true*?

1. In C++11, all classes can be used as base classes.
2. In C++11, only classes that are not declared as final can be used as base classes.
3. In C++11, only classes that are declared as base can be used as base classes.
4. None of the above

**ANS: b. In C++11, only classes that are not declared as final can be used as base classes.**

**Section 20.4 Type Fields and switch Statements**

20.4 Q1: Problems using switch logic to deal with many objects of different types do *not* include:

1. Forgetting to include an object in one of the cases.
2. Having to update the switch statement whenever a new type of object is added.
3. Having to track down every switch statement to do an update of object types.
4. Not being able to implement separate functions on different objects.

**ANS: d. Not being able to implement separate functions on different objects.**

**Section 20.5 Abstract Classes and Pure virtual Functions**

20.5 Q1: The line:

virtual double earnings() const = 0;

appears in a class definition. You *cannot* deduce that:

1. All classes that directly inherit from this class will *override* this method.
2. This class is an abstract class.
3. Any concrete class derived from this class will have an earnings function.
4. This class will probably be used as a base class for other classes.

**ANS: a. All classes that directly inherit from this class will override this method.**

20.5 Q2: Abstract classes:

1. Contain at *most* one pure virtual function.
2. Can have objects instantiated from them if the proper permissions are set.
3. *Cannot* have abstract derived classes.
4. Are defined, but the programmer *never* intends to instantiate any objects from them.

**ANS: d. Are defined, but the programmer *never* intends to instantiate any objects from them.**

20.5 Q3: The main difference between a pure virtual function and a virtual function is:

1. The return type.
2. The member access specifier.
3. That a pure virtual function cannot have an implementation.
4. The location in the class.

**ANS: c. That a pure virtual function cannot have an implementation.**

20.5 Q4: Which of the following is *not* allowed?

1. Objects of abstract classes.
2. Multiple pure virtual functions in a single abstract class.
3. References to abstract classes.
4. Arrays of pointers to abstract classes.

**ANS: a. Objects of abstract classes.**

**Section 20.6 Case Study: Payroll System Using Polymorphism**

20.6 Q1: What mistake prevents the following class declaration from functioning properly as an *abstract* class?

class Shape

{

public:

virtual double print() const;

double area() const { return base \* height; }

private:

double base;

double height;

};

1. There are no pure virtual functions.
2. There is a non-virtual function.
3. private variables are being accessed by a public function.
4. Nothing, it functions fine as an abstract class.

**ANS: a. There are no pure virtual functions.**

**Section 20.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”**

20.7 Q1: An abstract class will:

1. Have all zero function pointers in its *vtable*.
2. Have at least one zero function pointer in its *vtable*.
3. Share a *vtable* with a derived class.
4. Have fewer zero function pointers in its *vtable* than concrete classes have.

**ANS: b. Have at least one 0 in its *vtable*.**

20.7 Q2: Concrete classes that inherit virtual functions but do *not* override their implementations:

1. Have *vtables* which are the same as those of their base classes.
2. Receive their own copies of the virtual functions.
3. Receive pointers to their base classes’ virtual functions.
4. Receive pointers to pure virtual functions.

**ANS: c. Receive pointers to their base classes' virtual functions.**

20.7 Q3: The C++ compiler makes objects take up more space in memory if they:

1. Are derived from base classes.
2. Have virtual functions.
3. Have only protected members.
4. Are referenced by pointers.

**ANS: b. Have virtual functions.**

20.7 Q4: Abstract classes do *not* necessarily have:

1. A 0 pointer in their *vtable*.
2. A virtual function prototype with the notation = 0.
3. Zero instances of their class.
4. Zero references to their class.

**ANS: d. Zero references to their class.**

**Section 20.8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, dynamic\_cast, typeid and type\_info**

20.8 Q1: The line:

virtual double functionX() const = 0;

in a class definition indicates that the class is probably a:

1. Base class.
2. Derived class.
3. Protected class.
4. Library class.

**ANS: a. Base class.**

20.8 Q2: Run-time type information can be used to determine:

1. A function’s return type.
2. A function’s argument type.
3. An object’s type.
4. The number of arguments a function takes.

**ANS: c. An object’s type.**

20.8 Q3: The \_\_\_\_\_\_\_\_\_\_ operator returns a reference to a \_\_\_\_\_\_\_\_\_\_ object:

1. typeid, type\_info
2. typeinfo, type\_id
3. typeid, data\_type
4. typeinfo, type

**ANS: a. typeid, type\_info**

20.8 Q4: dynamic\_cast is often used to:

1. Perform type checking for objects.
2. Convert pointers to strings.
3. Upcast pointers.
4. Downcast pointers.

**ANS: d. Downcast pointers.**

**Section 20.9 Virtual Destructors**

20.9 Q1: virtual destructors *must* be used when:

1. The constructor in the base class is virtual.
2. delete is used on a base-class pointer to a derived-class object.
3. delete is used on a derived-class object.
4. A constructor in either the base class or derived class is virtual.

**ANS: b. delete is used on a base-class pointer to a derived-class object.**