Question 1.

Section 13.1 Introduction

13.1 Q1: Polymorphism is implemented via:

Non-virtual functions.

Member functions.

inline functions.

virtual functions and dynamic binding.

End of Question 1

virtual functions and dynamic binding.

Question 2.

Section 13.2 Base Classes and Derived Classes

13.2 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?

eat.

sleep.

lapWings.

move.

End of Question 2

lapWings

Question 3.

Section 13.3.1 Invoking Base-Class Functions from Derived-Class Objects

13.3.1 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();

It would depend on the implementation of the print function.

Yes, if print is a static function.

No.

Yes.

End of Question 3

Yes.

Question 4.

Section 13.3.2 Aiming Derived-Class Pointers at Base-Class Objects

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

Assigning the address of a derived-class object to a base-class pointer.

Assigning the address of a base-class object to a derived-class pointer.

Assigning the address of a derived-class object to a derived-class pointer.

Assigning the address of a base-class object to a base-class pointer.

End of Question 4

|  |  |
| --- | --- |
|  | Assigning the address of a base-class object to a derived-class pointer. |

Question 5.

Section 13.3.3 Derived-Class Member-Function Calls via Base-Class Pointers

13.3.3 Q1: Downcasting enables:

Making a base-class pointer into a derived-class pointer.

A derived-class object to be treated as a base-class object.

A base-class object to be treated as a derived-class object.

Making a derived-class pointer into a base -class pointer.

End of Question 5

Making a base-class pointer into a derived-class pointer.

Question 6.

Section 13.3.4 Virtual Functions

13.3.4 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:

private.

virtual.

friend.

protected.

End of Question 6

virtual.

Question 7.

13.3.4 Q2: Assume we have a base class Shape and derived classes Triangle and Rectangle. Which of the following member functions should be virtual?

isSquare.

isEquilateral.

isIsosceles.

isRegular.

End of Question 7

isRegular.

Question 8.

13.3.4 Q3: virtual functions must:

Be declared virtual in the base class.

Be declared virtual in every derived class.

Have the same implementation in every derived class.

Be overridden in every derived class.

End of Question 8

Be declared virtual in the base class.

Question 9.

13.3.4 Q4: Which of the following statements about virtual functions is false?

They can use either static or dynamic binding, depending on the handles on which the functions are called.

They allow the program to select the correct implementation at execution time.

They do not remain virtual down the inheritance hierarchy.

They can be called using the dot operator.

End of Question 9

|  |  |
| --- | --- |
|  | They do not remain virtual down the inheritance hierarchy. |

Question 10.

Section 13.4 Type Fields and switch Statements

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

Having to track down every switch statement to do an update of object types.

Not being able to implement separate functions on different objects.

Having to update the switch statement whenever a new type of object is added.

Forgetting to include an object in one of the cases.

End of Question 10

Not being able to implement separate functions on different objects.

Question 11.

Section 13.5 Abstract Classes and Pure virtual Functions

13.5 Q1: The line:

virtual double earnings() const = 0;

appears in a class definition. You cannot deduce that:

All classes that directly inherit from this class will override this method.

This class is an abstract class.

Any concrete class derived from this class will have an earnings function.

This class will probably be used as a base class for other classes.

End of Question 11

All classes that directly inherit from this class will override this method.

Question 12.

13.5 Q2: Abstract classes:

Can have objects instantiated from them if the proper permissions are set.

Contain at most one pure virtual function.

Cannot have abstract derived classes.

Are defined, but the programmer never intends to instantiate any objects from them.

End of Question 12

Are defined, but the programmer never intends to instantiate any objects from them.

Question 13.

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

The return type.

b. The member access specifier.

That a pure virtual function cannot have an implementation.

The location in the class.

End of Question 13

That a pure virtual function cannot have an implementation.

Question 14.

13.5 Q4: Which of the following is not allowed?

Arrays of pointers to abstract classes.

c. References to abstract classes.

Objects of abstract classes.

Multiple pure virtual functions in a single abstract class.

End of Question 14

Objects of abstract classes.

Question 15.

Section 13.6 Case Study: Payroll System Using Polymorphism

13.6 Q1: What mistakes 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;

};

There is a non-virtual function.

There are no pure virtual functions.

private variables are being accessed by a public function.

Nothing, it functions fine as an abstract class.

End of Question 15

There are no pure virtual functions.

Question 16.

Section 13.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”

13.7 Q1: An abstract class will:

Have at least one 0 in its vtable.

Have all 0's in its vtable.

Share a vtable with a derived class.

Have fewer 0's in its vtable than concrete classes have.

End of Question 16

Have at least one 0 in its *vtable*.

Question 17.

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

Receive pointers to their base classes’ virtual functions.

Receive pointers to pure virtual functions.

Have vtables which are the same as those of their base classes.

Receive their own copies of the virtual functions.

End of Question 17

Receive pointers to their base classes’ virtual functions.

Question 18.

13.7 Q3: Polymorphism and virtual functions are not appropriate for:

d. Programs that use many inherited classes with similar functions.

Programs that must be easily extensible.

Programs where classes may be added in the future.

Programs that have strict memory and processor requirements.

End of Question 18

Programs that have strict memory and processor requirements.

Question 19.

13.7 Q4: The C++ compiler makes objects take up more space in memory if they:

Have only protected members.

Are derived from base classes.

Have virtual functions.

Are referenced by pointers.

End of Question 19

Have virtual functions.

Question 20.

13.7 Q5: Abstract classes do not necessarily have:

Zero references to their class.

A virtual function prototype with the notation = 0.

A 0 pointer in their vtable.

Zero instances of their class.

End of Question 20

Zero references to their class.

Question 21.

13.7 Q6: Which statement is not true about dynamic binding?

It allows software developers to derive new classes compatible with existing software.

It eliminates the usefulness of separate header and source files.

It requires additional overhead when the program is executing.

The program chooses the correct functions at execution time, rather than at compile time.

End of Question 21

It eliminates the usefulness of separate header and source files.

Question 22.

Section 13.8 Case Study: Payroll System Using Polymorphism and Run-Time Type Information with downcasting, dynamic\_cast, typeid and type\_info

13.8 Q1: The line:

virtual double functionX() const = 0; in a class definition indicates that the class is probably a:

Protected class.

Derived class.

Library class.

Base class.

End of Question 22

Base class.

Question 23.

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

A function’s argument type.

An object’s type.

A function’s return type.

The number of arguments a function takes.

End of Question 23

An object’s type.

Question 24.

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

typeinfo, type.

typeinfo, type\_id.

typeid, type\_info.

typeid, data\_type.

End of Question 24

typeid, type\_info.

Question 25.

13.8 Q4: dynamic\_cast is often used to:

Downcast pointers.

Upcast pointers.

Convert pointers to strings.

Perform type checking for objects.

End of Question 25

Downcast pointers.

Question 26.

Section 13.9 Virtual Destructors

13.9 Q1: virtual destructors must be used when:

delete is used on a base-class pointer to a derived-class object.

The constructor in the base class is virtual.

delete is used on a derived-class object.

A constructor in either the base class or derived class is virtual.

End of Question 26

delete is used on a base-class pointer to a derived-class object.

Question 27.

Section 13.10 (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System

13.10 Q1: Which attribute or behavior would we not factor out of the Pants and Socks classes and into the Clothing base class?

numberOfPockets.

isClean.

material.

color.

End of Question 27

numberOfPockets.