96. What is the purpose of type erasure in C++ polymorphism?

97. What is an adapter pattern in C++ polymorphism?

98. How do you implement an adapter pattern in C++ polymorphism?

99. What is the purpose of an adapter pattern in C++ polymorphism?

100. What is the difference between dynamic polymorphism and static polymorphism in C++?

I hope this list helps you practice and deepen your understanding of polymorphism in C++!

TABLE OF CONTENTS

1. What is polymorphism in C++?

Polymorphism in C++ is the ability of an object to take on many forms. It allows objects of different classes to be treated as objects of a common base class during runtime, enabling the use of a single interface to represent multiple types.

2. What is the purpose of polymorphism in object-oriented programming?

The purpose of polymorphism in object-oriented programming is to improve code flexibility, reusability, and extensibility. It allows for code that can work with objects of different classes through a common interface, promoting code organization, modularity, and simplifying code maintenance.

3. How do you achieve polymorphism in C++?

Polymorphism in C++ can be achieved through inheritance and virtual functions. By defining a common base class and using virtual functions, different derived classes can override those functions to provide their specific implementation. The objects of derived classes can be treated as objects of the base class, enabling polymorphic behavior.

4. What is runtime polymorphism in C++?

Runtime polymorphism, also known as dynamic polymorphism or late binding, occurs when the function call is resolved at runtime based on the actual type of the object. It allows different derived classes to provide their own implementation of a function defined in the base class, and the appropriate function is called based on the type of the object being referred to.

5. How do you achieve runtime polymorphism in C++?

Runtime polymorphism in C++ is achieved by using virtual functions. Declare a base class function as virtual, and override it in the derived classes. When the function is called through a base class pointer or reference, the derived class's implementation is invoked based on the actual type of the object being referred to.

6. What is static polymorphism in C++?

Static polymorphism, also known as compile-time polymorphism or early binding, is resolved at compile-time rather than runtime. It is achieved through function overloading and templates in C++. The compiler selects the appropriate function or template instantiation based on the types and number of arguments at compile-time.

7. How do you achieve static polymorphism in C++?

Static polymorphism in C++ is achieved through function overloading and templates. Function overloading allows multiple functions with the same name but different parameters. The appropriate function is resolved at compile-time based on the arguments provided. Templates allow writing generic code that can work with different data types, and the compiler generates specific code for each instantiation.

8. What is function overloading in C++?

9. How do you overload functions in C++?

To overload functions in C++, you define multiple functions with the same name but different parameter lists. The parameter lists can differ in terms of the number of parameters, the types of parameters, or both. The functions must have the same return type or differ in their return types, as the return type alone is not sufficient to resolve the function call.

10. What is the difference between function overloading and function overriding in C++?

Function overloading is a static polymorphism mechanism that allows multiple functions with the same name but different parameters in the same scope. The appropriate function is resolved at compile-time based on the arguments provided.

Function overriding, on the other hand, is a runtime polymorphism mechanism that occurs when a derived class provides its own implementation of a function that is already defined in the base class. The appropriate function is resolved at runtime based on the actual type of the object being referred to.

11. What is function overriding in C++?

Function overriding in C++ is the process of providing a different implementation of a base class function in a derived class. The derived class function must have the same name, return type, and parameters as the base class function. The function is marked with the `override` keyword to indicate that it is intended to override a base class function.

12. How do you override a base class function in a derived class in C++?

To override a base class function in a derived class in C++, you declare a function with the same name, return type, and parameters as the base class function in the derived class. Add the `override` keyword at the end of the function declaration to explicitly indicate that it is intended to override a base class function.

13. What is the difference between virtual functions and overridden functions in C++?

A virtual function is a function declared in a base class with the `virtual` keyword, which allows it to be overridden in derived classes. It enables runtime polymorphism by ensuring that the appropriate derived class function is called based on the actual type of the object.

An overridden function is a function in a derived class that provides a different implementation than the base class function with the same name, return type, and parameters. It is marked with the `override` keyword to indicate that it intends to override the base class function.

14. What is the use of the ‘override’ keyword in C++?

The `override` keyword in C++ is used to indicate that a function in a derived class is intended to override a base class function. It helps catch potential errors at compile-time by ensuring that the function signature matches the base class function it is supposed to override. If the function doesn’t match any base class function, a compilation error occurs.

15. How do you use the ‘override’ keyword in a derived class function in C++?

To use the `override` keyword in a derived class function in C++, simply add it at the end of the function declaration, after the parameter list and before the function body. The presence of the `override` keyword indicates that the function is intended to override a base class function.

16. What is a pure virtual function in C++?

A pure virtual function in C++ is a virtual function declared in a base class but does not provide any implementation in the base class. It is denoted by appending `= 0` to the function declaration. A class

```cpp
virtual void pureVirtualFunction() = 0;
```

To define a pure virtual function, you need to provide the implementation in the derived class. The derived class must override the pure virtual function and provide a body for it.

18. Can a pure virtual function have a definition in C++?

No, a pure virtual function cannot have a definition in C++. The pure virtual function declared in the base class does not provide any implementation, and it is the responsibility of the derived classes to override and provide the implementation. Only non-pure virtual functions can have definitions.

19. What is the purpose of pure virtual functions in C++?

The purpose of pure virtual functions in C++ is to create an interface or a contract that derived classes must adhere to. It allows the base class to define a common set of functions that derived classes are required to implement. Pure virtual functions enable runtime polymorphism and dynamic binding by allowing objects of derived classes to be treated as objects of the base class.

20. Can you create objects of a class with pure virtual functions in C++?

No, you cannot create objects of a class with pure virtual functions in C++. A class that contains at least one pure virtual function is considered an abstract class, and abstract classes cannot be instantiated. They serve as base classes for derived classes that provide implementations for the pure virtual functions.

Sure! Here are the answers to the remaining questions:

21. What is an abstract class in C++?

An abstract class in C++ is a class that contains at least one pure virtual function. It serves as a base class for derived classes and defines an interface that derived classes must implement. Abstract classes cannot be instantiated, but they provide a common structure and behavior that derived classes can inherit and extend.

22. How do you define and use an abstract class in C++?

To define an abstract class in C++, you declare at least one pure virtual function in the class. This is done by appending `= 0` to the function declaration. For example:

```cpp
class AbstractClass {
public:
virtual void pureVirtualFunction() = 0;
// Other member functions and data members
};
```

To use an abstract class, you derive a new class from it and provide implementations for its pure virtual functions. The derived class becomes a concrete class that can be instantiated.

23. Can you create objects of an abstract class in C++?

No, you cannot create objects of an abstract class in C++. Since an abstract class contains at least one pure virtual function, it is considered incomplete and cannot be instantiated. However, you can create objects of derived classes that inherit from the abstract class.

24. What is a virtual function in C++?

25. How do you declare and define a virtual function in C++?

To declare a virtual function in C++, you simply include the `virtual` keyword before the return type in the function declaration in the base class. For example:

```cpp
class Base {
public:
virtual void virtualFunction();
// Other member functions and data members
};
```

To define a virtual function, you provide its implementation in the base class or in any derived class that overrides it. The derived classes use the `override` keyword to indicate that they are overriding the virtual function.

26. What is the purpose of a virtual function in C++?

The purpose of a virtual function in C++ is to enable polymorphism and dynamic binding. By declaring a function as virtual in the base class, you allow derived classes to provide their own implementations of the function. This enables the appropriate function to be called based on the actual type of the object, even when the object is accessed through a pointer or reference of the base class.

27. How do you override a virtual function in a derived class in C++?

To override a virtual function in a derived class in C++, you declare a function with the same name, return type, and parameters as the virtual function in the base class. Add the `override` keyword at the end of the function declaration to explicitly indicate that it is intended to override the base class function.

```cpp
class Derived : public Base {
public:
void virtualFunction() override;
// Other member functions and data members
};
```

28. What is the difference between virtual functions and non-virtual functions in C++?

The main difference between virtual functions and non-virtual functions in C++ is how they are resolved at runtime. Non-virtual functions are resolved at compile-time based on the type of the pointer or reference, while virtual functions are resolved at runtime based on the type of the actual object being pointed to or referenced.

When a non-virtual function is called through a pointer or reference of the base class, the function in the base class is always called, regardless of the actual type of the object. In contrast, when

a virtual function is called through a pointer or reference of the base class, the appropriate function in the derived class is called based on the actual type of the object.

29. What is a pure virtual destructor in C++?

A pure virtual destructor in C++ is a virtual destructor that is declared as pure virtual in the base class. It is declared using the `= 0` syntax, similar to pure virtual functions. A class with a pure virtual destructor is considered an abstract class and cannot be instantiated.

30. How do you declare and define a pure virtual destructor in C++?

```cpp
class AbstractClass {
public:
virtual ~AbstractClass() = 0;
// Other member functions and data members
};

// Define the pure virtual destructor
AbstractClass::~AbstractClass() {}
```

Note that the pure virtual destructor still needs a definition, which can be provided outside the class declaration.

31. Can a pure virtual destructor have a definition in C++?

Yes, a pure virtual destructor can have a definition in C++. Although the destructor is declared as pure virtual in the base class, it still needs to have a definition to satisfy the linker. The definition can be provided outside the class declaration.

```cpp
AbstractClass::~AbstractClass() {}
```

32. What is the purpose of pure virtual destructors in C++?

The purpose of pure virtual destructors in C++ is to ensure that derived classes properly clean up their resources when objects are destroyed. By making the destructor pure virtual in the base class, derived classes are required to provide their own implementations of the destructor. This ensures that the appropriate derived class destructor is called when an object is destroyed through a pointer or reference of the base class type.

32. What is the purpose of pure virtual destructors in C++?

The purpose of pure virtual destructors in C++ is to ensure that derived classes properly clean up their resources when objects are destroyed. By making the destructor pure virtual in the base class, derived classes are required to provide their own implementations of the destructor. This ensures that the appropriate derived class destructor is called when an object is destroyed through a pointer or reference of the base class type.

33. What is an interface in C++?

In C++, an interface is a class that contains only pure virtual functions and no data members or non-virtual member functions. It defines a contract or set of methods that derived classes must implement. Interfaces provide a way to achieve pure abstract behavior in C++, similar to other programming languages.

34. How do you define and use an interface in C++?

To define an interface in C++, you create a class with only pure virtual functions and no data members or non-virtual member functions. For example:

```cpp
class Interface {
public:
virtual void method1() = 0;
virtual void method2() = 0;
// Other pure virtual functions
```

35. Can a class have multiple interfaces in C++?

Yes, a class in C++ can inherit from multiple interfaces. C++ supports multiple inheritance, which allows a class to inherit from multiple base classes, including interfaces. By inheriting from multiple interfaces, a class can implement the contracts defined by each interface and provide the required functionality.

36. What is the difference between an abstract class and an interface in C++?

The main difference between an abstract class and an interface in C++ lies in their implementation and usage. An abstract class can contain a mixture of pure virtual functions, regular member functions, and data members. It can also have defined member functions and constructors. In contrast, an interface contains only pure virtual functions and has no data members or non-virtual member functions.

An abstract class can provide default implementations for some functions and can be used as a base class for derived classes. It may have some concrete functionality in addition to the pure virtual functions. On the other hand, an interface is purely a contract and provides no implementation. Classes implementing an interface must provide the implementation for all the pure virtual functions defined in the interface.

37. What is the role of the vtable in C++ polymorphism?

The vtable (virtual function table) is a mechanism used by C++ compilers to implement dynamic dispatch and achieve polymorphism. It is a data structure associated with each class that contains virtual functions. The vtable stores the addresses of the virtual functions for that class.

When an object is accessed through a pointer or reference of a base class type, and a virtual function is called, the compiler uses the vtable to determine the appropriate function to call based on the actual type of the object. The vtable allows the correct virtual function to be invoked, even when dealing with objects of different derived classes that have overridden the virtual function.

38. How does the vtable work in C++ polymorphism?

The vtable in C++ works by providing a mapping between the virtual function calls and their actual implementations in derived classes. Each class that contains at least one virtual function has a corresponding vtable. The vtable is a static data structure that is created at compile-time.

The vtable is organized as an array of function pointers, where each entry represents a virtual function in the class. When an object is created, a hidden pointer called the vpointer (or vptr) is added to the object's memory layout. This vpointer points to the

vtable associated with the class.

When a virtual function is called on an object, the compiler uses the vpointer to access the appropriate entry in the vtable, which contains the address of the function to be called. This allows the correct virtual function implementation to be invoked based on the actual type of the object, achieving dynamic dispatch and polymorphism.

39. What is a virtual base class in C++ polymorphism?

A virtual base class is a class that is declared as virtual when it is used as a base class in multiple inheritance scenarios. It is used to prevent the creation of multiple instances of the base class when there are multiple paths to reach it in the inheritance hierarchy.

When a class is declared as a virtual base class, only one instance of its data members is shared among all the derived classes that inherit from it. This ensures that there are no duplicate instances of the virtual base class, resolving issues related to ambiguity and data redundancy in multiple inheritance.

40. How do you declare a virtual base class in C++ polymorphism?

To declare a virtual base class in C++, you add the `virtual` keyword before the base class name in the derived class declaration. For example:

```
};

class DerivedClass : virtual public BaseClass {
// Derived class members
};
```

The `virtual` keyword indicates that the `BaseClass` should be treated as a virtual base class. This ensures that only one instance of `BaseClass` is shared among all the derived classes that inherit from it, regardless of the number of paths leading to it in the inheritance hierarchy.

41. What is the purpose of a virtual base class in C++ polymorphism?

The purpose of a virtual base class in C++ polymorphism is to prevent the duplication of inherited members when multiple paths in an inheritance hierarchy lead to the same base class. It ensures that only one instance of the virtual base class is shared among the derived classes, avoiding issues of ambiguity and redundant data.

By declaring a base class as virtual, you eliminate the creation of duplicate base class subobjects and ensure that the derived classes can access the shared base class members correctly.

42. What is a pure abstract class in C++ polymorphism?

A pure abstract class, also known as an interface class, is a class that contains only pure virtual functions and no data members or non-virtual member functions. It defines a contract or a set of methods that derived classes must implement. A pure abstract class cannot be instantiated; it is meant to be used as a base class for other classes that provide concrete implementations of the pure virtual functions.

43. How do you define and use a pure abstract class in C++ polymorphism?

To define a pure abstract class in C++, you declare a class with only pure virtual functions and no data members or non-virtual member functions. For example:

```cpp
class PureAbstractClass {
public:
virtual void method1() = 0;
virtual void method2() = 0;
// Other pure virtual functions
};
```

To use a pure abstract class, you derive a class from it and provide concrete implementations for all the pure virtual functions defined in the base class. The derived class becomes a concrete class that can be instantiated and used.

44. Can you create objects of a pure abstract class in C++ polymorphism?

No, you cannot create objects of a pure abstract class in C++ polymorphism. A pure abstract class is an incomplete class that contains pure virtual functions and cannot be instantiated directly. Its purpose is to serve as a base class for derived classes that provide concrete implementations of the pure virtual functions.

45. What is the difference between a virtual function and a pure virtual function in C++ polymorphism?

A pure virtual function, on the other hand, is a virtual function declared in a base class using the `virtual` keyword and set to 0 (pure virtual function syntax). It is a function that has no implementation in the base class and must be overridden by any derived class that intends to be instantiated. Classes containing pure virtual functions are abstract classes and cannot be instantiated directly.

46. What is an object pointer in C++ polymorphism?

An object pointer in C++ polymorphism is a pointer variable that can hold the memory address of an object. It allows accessing and manipulating objects dynamically at runtime. Object pointers are commonly used to achieve polymorphic behavior, where a pointer of a base class type can point to objects of derived classes.

47. How do you declare and initialize an object pointer in C++ polymorphism?

To declare an object pointer in C++ polymorphism, you specify the pointer type using the class name followed by an asterisk (*). For example:

```cpp
ClassName* ptr;
```

To initialize an object pointer, you assign it the memory address of an existing object using the address-of operator (&) or through dynamic memory allocation using the new operator. For example:

```cpp
ClassName obj;
ClassName* ptr = &obj;

// or

ClassName* ptr = new ClassName();
```

48. How do you use an object pointer to achieve polymorphism in C++?

To achieve polymorphism in C

++ using an object pointer, you can create a pointer of the base class type and assign it the address of an object of the derived class. This allows you to access the derived class's members through the base class pointer using virtual functions.

Here's an example:

```cpp
class BaseClass {
public:
virtual void polymorphicFunction() {
// Base class implementation
}
};

class DerivedClass : public BaseClass {
public:
void polymorphicFunction() override {
// Derived class implementation
```

```
ptr->polymorphicFunction(); // Calls the derived class implementation
delete ptr;
return 0;
}
```

In this example, the `BaseClass` pointer `ptr` is assigned the address of a `DerivedClass` object. When calling the `polymorphicFunction()` through the pointer, the derived class implementation is invoked instead of the base class implementation due to dynamic dispatch and polymorphism.

49. What is a base class pointer in C++ polymorphism?

A base class pointer in C++ polymorphism is a pointer variable of the base class type that can hold the address of objects of either the base class or any of its derived classes. It provides a way to refer to objects of different classes in a polymorphic manner.

50. How do you declare and initialize a base class pointer in C++ polymorphism?

To declare a base class pointer in C++ polymorphism, you specify the pointer type using the base class name followed by an asterisk (*). For example:

```cpp
BaseClassName* ptr;
```

To initialize a base class pointer, you assign it the address of an object of the base class or any of its derived classes. For example:

```cpp
BaseClassName obj;
BaseClassName* ptr = &obj;

// or

DerivedClassName obj;
BaseClassName* ptr = &obj;
```

In both cases, the base class pointer `ptr` can hold the address of either a `BaseClassName` object or a `DerivedClassName` object, allowing polymorphic behavior.

51. How do you use a base class pointer to achieve polymorphism in C++?

You can use a base class pointer to achieve polymorphism in C++ by assigning it the address of an object of the derived class. Through the base class pointer, you can access member functions and overridden virtual functions of the derived class, allowing dynamic dispatch and runtime determination of the appropriate function to call.

Here’s an example:

```cpp
class BaseClass {
public:
virtual void polymorphicFunction() {
// Base class implementation
```

```
void polymorphicFunction() override {
// Derived class implementation
}
};

int main() {
BaseClass* ptr;
DerivedClass obj;
ptr = &obj;
ptr->polymorphicFunction(); // Calls the derived class implementation
return 0;
}
```

In this example, the base class pointer `ptr` is assigned the address of a `DerivedClass` object. When calling the `polymorphicFunction()` through the pointer, the derived class implementation is invoked instead of the base class implementation due to dynamic dispatch and polymorphism.

52. What is the difference between an object pointer and a base class pointer in C++ polymorphism?

An object pointer in C++ polymorphism is a pointer that holds the address of a specific object, allowing access to the members of that object. It points directly to the memory location of the object.

A base class pointer, on the other hand, is a pointer of the base class type that can hold the address of objects of the base class or any of its derived classes. It allows accessing and manipulating objects dynamically at runtime, supporting polymorphism.

The key difference is that an object pointer points directly to an object of a specific class, while a base class pointer can point to objects of multiple classes in an inheritance hierarchy, providing polymorphic behavior.

53. What is a virtual destructor in C++ polymorphism?

A virtual destructor in C++ polymorphism is a destructor declared in a base class using the `virtual` keyword. It enables the proper destruction of derived class objects through a base class pointer. When a derived class object is deleted via a base class pointer, the virtual destructor ensures that the destructor of the derived class is called in addition to the base class destructor.

54. How do you declare and define a virtual destructor in C++ polymorphism?

To declare a virtual destructor in C++ polymorphism, you add the `virtual` keyword before the destructor declaration in the base class. For example:

```cpp
class BaseClass {
public:
virtual ~BaseClass() {
// Destructor implementation
}
};
```

To define the virtual destructor, you provide the implementation of the destructor within the class definition or in a separate implementation file.

55. What is the purpose of a virtual destructor in C++ polymorphism?

The purpose of a virtual destructor in C++ polymorphism is to ensure proper destruction of derived class

Without a virtual destructor, only the base class destructor would be called, leading to potential resource leaks and undefined behavior when deleting derived class objects through a base class pointer.

56. What is slicing in C++ polymorphism?

Slicing in C++ polymorphism refers to the loss of derived class-specific information when an object of a derived class is assigned or passed by value to an object of the base class. It occurs when a derived class object is treated as an object of its base class type, resulting in the base class portion of the derived

object being copied or assigned, while the derived class-specific data is sliced off.

57. How does slicing occur in C++ polymorphism?

Slicing occurs in C++ polymorphism when an object of a derived class is assigned or passed by value to an object of the base class. During the assignment or copy, only the base class portion of the derived object is copied, while the derived class-specific data is lost. This loss of derived class information is known as slicing.

For example:

```cpp
class BaseClass {
public:
int baseData;
};

class DerivedClass : public BaseClass {
public:
int derivedData;
};

int main() {
DerivedClass derivedObj;
derivedObj.baseData = 10;
derivedObj.derivedData = 20;

BaseClass baseObj = derivedObj; // Slicing occurs here
// Only the baseData is copied, derivedData is lost

return 0;
}
```

In this example, assigning the `derivedObj` of type `DerivedClass` to the `baseObj` of type `BaseClass` results in slicing. Only the `baseData` member is copied, and the `derivedData` member is lost.

58. What is dynamic binding in C++ polymorphism?

Dynamic binding, also known as late binding or runtime binding, is a feature of C++ polymorphism that allows the selection of the appropriate function implementation at runtime based on the actual object type. It enables the invocation of overridden virtual functions of derived classes through a base class pointer or reference.

With dynamic binding, the decision on which function implementation to call is made at runtime based on the actual type of the object, rather than the static (compile-time) type of the pointer or reference.

59. How do you achieve dynamic binding in C++ polymorphism?

Dynamic binding in C++ polymorphism is achieved by using virtual functions. To enable dynamic

60. What is the difference between early binding and late binding in C++ polymorphism?

The difference between early binding and late binding in C++ polymorphism is as follows:

– Early binding, also known as static binding or compile-time binding, is when the binding between a function call and its implementation is resolved at compile-time based on the static (known) type of the object or reference. It occurs for non-virtual functions and functions called directly on objects, rather than through pointers or references. Early binding is resolved using the static type information available during compilation.

– Late binding, also known as dynamic binding or runtime binding, is when the binding between a function call and its implementation is deferred until runtime. It occurs for virtual functions called through a base class pointer or reference. Late binding is resolved using the dynamic (actual) type of the object determined at runtime.

The key difference is that early binding is resolved at compile-time based on the static type, while late binding is resolved at runtime based on the actual type of the object.

61. What is the ‘final’ keyword in C++ polymorphism?

The ‘final’ keyword in C++ polymorphism is used to prevent further overriding of a virtual function or inheriting from a class. It is an attribute that can be applied to virtual functions and classes to indicate that they are not intended to be overridden or inherited, respectively.

When a virtual function is declared as ‘final’, it cannot be overridden in any derived class. Similarly, when a class is marked as ‘final’, it cannot be inherited

by any other class.

62. How do you use the ‘final’ keyword to prevent function overriding in C++ polymorphism?

To prevent function overriding using the ‘final’ keyword in C++ polymorphism, you declare the virtual function with the ‘final’ specifier in the base class. This indicates that the function should not be further overridden in any derived class.

Here’s an example:

```cpp
class BaseClass {
public:
virtual void foo() final {
// Function implementation
}
};

class DerivedClass : public BaseClass {
public:
void foo() override { // Error: Cannot override final function
// Function implementation
}
};
```

In this example, the ‘foo()’ function in the ‘BaseClass’ is declared as ‘final’, preventing any derived class, such as ‘DerivedClass’, from overriding it. If an attempt is made to override the function in a derived class, a compilation error will occur.

The ‘final’ keyword helps in enforcing design decisions and preventing unintended modifications or extensions that could break the intended behavior or contract of a class or function.

64. What is an abstract base class in C++ polymorphism?

An abstract base class in C++ polymorphism is a class that is designed to be inherited from, but cannot be instantiated on its own. It serves as an interface or blueprint for derived classes to define common behavior or provide a common interface.

An abstract base class typically contains one or more pure virtual functions, making it an abstract class. A class becomes abstract when it declares at least one pure virtual function, denoted by the ‘= 0’ syntax, which means the function has no implementation in the base class.

65. How do you define and use an abstract base class in C++ polymorphism?

To define an abstract base class in C++ polymorphism, you declare one or more pure virtual functions in the base class. A pure virtual function is declared using the syntax `virtual void functionName() = 0;` without providing an implementation in the base class.

Here’s an example:

```cpp
class AbstractBase {
public:
virtual void pureVirtualFunction() = 0;
};

class DerivedClass : public AbstractBase {
public:
void pureVirtualFunction() override {
// Implementation in the derived class
}
};

int main() {
AbstractBase* ptr = new DerivedClass();
ptr->pureVirtualFunction(); // Calls the derived class implementation
delete ptr;
return 0;
}
```

In this example, the `AbstractBase` class is an abstract base class with a pure virtual function `pureVirtualFunction()`. The `DerivedClass` inherits from the `AbstractBase` and provides an implementation for the pure virtual function. An object of the derived class is created using a base class pointer, and the pure virtual function is called through the pointer, resulting in dynamic dispatch and invocation of the derived class implementation.

66. Can you create objects of an abstract base class in C++ polymorphism?

No, you cannot create objects of an abstract base class in C++ polymorphism. An abstract base class is designed to be inherited from and serves as an interface or blueprint for derived classes. It contains pure virtual functions that must be overridden in the derived classes to provide concrete implementations.

Since the abstract base class contains pure virtual functions without implementations, it is considered incomplete and cannot be instantiated on its own. An attempt to create an

67. What is the difference between an abstract base class and a concrete base class in C++ polymorphism?

The difference between an abstract base class and a concrete base class in C++ polymorphism is as follows:

– An abstract base class is a class that is designed to be inherited from and cannot be instantiated on its own. It contains pure virtual functions, making it an abstract class. The purpose of an abstract base class is to provide a common interface or behavior that must be implemented by the derived classes.

– A concrete base class, on the other hand, is a class that can be instantiated and provides concrete implementations for all its member functions. It may also have virtual functions that can be overridden by the derived classes. The purpose of a concrete base class is to provide a base implementation that can be shared by multiple derived classes.

In summary, an abstract base class contains pure virtual functions and cannot be instantiated, while a concrete base class provides concrete implementations and can be instantiated.

68. What is a pure virtual base class in C++ polymorphism?

A pure virtual base class in C++ polymorphism is an abstract base class that serves as a common interface or blueprint for multiple related classes. It contains pure virtual functions that must be overridden by all the derived classes.

A pure virtual base class is used when you want to enforce derived classes to provide implementations for specific member functions. The derived classes are required to override all the pure virtual functions, making them concrete classes.

69. How do you define and use a pure virtual base class in C++ polymorphism?

To define a pure virtual base class in C++ polymorphism, you declare one or more pure virtual functions in the base class. A pure virtual function is declared using the syntax `virtual void functionName() = 0;` without providing an implementation in the base class.

Here's an example:

```cpp
class PureVirtualBase {
public:
virtual void pureVirtualFunction() = 0;
};

class DerivedClass : public PureVirtualBase {
public:
void pureVirtualFunction() override {
// Implementation in the derived class
}
};

int main() {
PureVirtualBase* ptr = new DerivedClass();
ptr->pureVirtualFunction(); // Calls the derived class implementation
delete ptr;
return 0;
}
```

In this example, the `PureVirtualBase` class is a pure virtual base class with a pure virtual function