Interview Exam: IModule Abstract Class - Answers

Candidate

March 13, 2025

Responses to Questions

1. **Abstract Classes**

   (a) **What is an abstract class in C++? How is it defined?**

   An abstract class in C++ is a class that cannot be instantiated on its own and is typically used as a base class for other classes. It is defined by having at least one pure virtual function, which is declared using the syntax 'virtual void function() = 0;'.

   (b) **Why is the `IModule` class declared as abstract? What is its purpose?**

   The `IModule` class is declared as abstract to define a common interface for all derived modules. Its purpose is to enforce that derived classes implement specific methods like `start()`, `stop()`, `receiveData()`, and `transitionToNextModule()`.

   (c) **Can you instantiate an object of an abstract class? Why or why not?**

   No, you cannot instantiate an object of an abstract class because it contains at least one pure virtual function, which makes the class incomplete. Abstract classes are meant to be inherited and implemented by derived classes.

2. **Virtual Functions**

(a) **What is a virtual function in C++? How is it different from a regular function?**

A virtual function is a member function that can be overridden in derived classes. It allows for dynamic dispatch, meaning the correct function implementation is determined at runtime based on the object's type. A regular function, on the other hand, is resolved at compile time.

(b) **What is a pure virtual function? How is it declared in the `IModule` class?**

A pure virtual function is a virtual function that has no implementation in the base class and must be overridden by derived classes. It is declared using the syntax 'virtual void function() = 0;'. In the `IModule` class, functions like `start()`, `stop()`, `receiveData()`, and `transitionToNextModule()` are pure virtual.

(c) **Can you have a non-virtual function in an abstract class? If so, what is its purpose?**

Yes, you can have non-virtual functions in an abstract class. These functions provide common behavior that can be shared by all derived classes. They are not meant to be overridden and are resolved at compile time.

3. **Class Design**

(a) **Why does the `IModule` class have a virtual destructor? What happens if a destructor is not declared as virtual in a base class?**

The `IModule` class has a virtual destructor to ensure that the destructor of the derived class is called when an object is deleted through a base class pointer. If the destructor is not virtual, only the base class destructor will be called, leading to potential memory leaks or undefined behavior.

(b) **What is the purpose of the `byteDataQueue`, `queueMutex`, and `queueLogMutex` members in the `IModule` class?**

The `byteDataQueue` is used to store incoming byte vectors. The `queueMutex` and `queueLogMutex` are used to protect access to shared resources, ensuring thread safety when multiple threads access the queue or log data.

(c) **Can you add non-pure virtual functions to the `IModule` class? What would be their role?**

Yes, you can add non-pure virtual functions to the `IModule` class. Their role would be to provide default behavior that derived classes can optionally override. This allows for flexibility in the implementation of derived classes.

4. **Inheritance and Polymorphism**

(a) **How does the `IModule` class support polymorphism? Provide an example.**

The `IModule` class supports polymorphism through its pure virtual functions. For example, you can have a pointer of type `IModule*` that points to an object of a derived class (e.g., `RandomByteGenerator`). When you call a virtual function like `start()`, the derived class's implementation is executed.

(b) **What happens if a derived class does not implement all the pure virtual functions of the `IModule` class?**

If a derived class does not implement all the pure virtual functions of the `IModule` class, it will also be considered an abstract class and cannot be instantiated.

(c) **Can you add new pure virtual functions to the `IModule` class without breaking existing derived classes? Why or why not?**

No, adding new pure virtual functions to the `IModule` class would break existing derived classes because they would no longer implement all the pure virtual functions, making them abstract as well.

5. **Error Handling**

(a) **How would you handle errors in the `IModule` class if a derived class fails to implement a required function?**

If a derived class fails to implement a required function, the program will not compile because the derived class will be considered abstract. To handle this, ensure that all derived classes implement the required functions.

(b) **What happens if a derived class does not properly manage the `queueMutex` or `queueLogMutex` members?**

If a derived class does not properly manage the mutexes, it could lead to race conditions, data corruption, or undefined behavior. Proper locking and unlocking mechanisms must be implemented in derived classes.

6. **Code Improvements**

(a) **Are there any potential issues with the current implementation of the `IModule` class? How would you improve it?**

Potential issues:

- The mutexes (`queueMutex` and `queueLogMutex`) are exposed in the base class, which could lead to improper usage in derived classes.
- The `byteDataQueue` is not protected by a mutex in the base class, which could lead to race conditions.

Improvements:

- Encapsulate the mutexes and queue in private members and provide thread-safe access methods.
- Add error handling for mutex operations.

(b) **How would you modify the `IModule` class to make it more flexible for future extensions?**

To make the `IModule` class more flexible:

- Add non-pure virtual functions with default implementations for common behavior.
- Use templates or interfaces to allow for different types of data processing.
- Provide hooks or callbacks for custom behavior in derived classes.

Conclusion

These responses provide a detailed explanation of the `IModule` abstract class and its functionality. The code demonstrates good practices in abstract class

design, but there is room for improvement in encapsulation and flexibility.