C++ Quiz: Responses

Responses

Q1. Output of the given ‘std::vector‘ code:

This code will cause undefined behavior because the iterator ‘it‘ becomes invalid after the call to ‘v.erase(it)‘. The correct way to implement this is as follows:

```
std::vector<int> v = {1, 2, 3};
for (auto it = v.begin(); it != v.end(); ) {
    if (*it == 2) {
        it = v.erase(it);
    } else {
        ++it;
    }
}
```

Q2. Function Overloading vs. Template Specialization:

Function overloading is resolved at compile time based on the function signature, while template specialization allows providing specific behavior for certain types. Example:

```
// Function overloading
void print(int x) { std::cout << "int: " << x; }
void print(double x) { std::cout << "double: " << x; }

// Template specialization
template <typename T>
void print(T x) { std::cout << x; }
template<>
void print<int>(int x) { std::cout << "Specialized int: " << x; }
```

Q3. Undefined behavior of the given code:

The issue is caused by the incorrect ‘delete‘ operation. Since ‘ptr‘ points to an array, ‘delete[] ptr‘ should be used instead of ‘delete ptr‘.

Q4. Key Differences Between ‘std::unique_ptr‘ and ‘std::shared_ptr‘:

- ‘std::unique_ptr‘ provides exclusive ownership and cannot be shared. - ‘std::shared_ptr‘ uses reference counting to allow multiple shared ownership.

Use cases: - Use ‘std::unique_ptr‘ for single-owner scenarios (e.g., managing a resource in a single class). - Use ‘std::shared_ptr‘ when ownership needs to be shared across multiple entities.

Q5. Output of the given vector code:

Output:

```
3 10
```

Explanation: - ‘v.size()‘ is 3 because the vector is initialized with three elements. - ‘v.capacity()‘ is 10 because the ‘reserve(10)‘ call pre-allocates space for up to 10 elements.

Q6. Lambda function to sort by second element:

```
std::vector<std::pair<int, int>> vec = { {1, 2}, {3, 1}, {2, 3} };
std::sort(vec.begin(), vec.end(), [](const auto &a, const auto &b) {
    return a.second > b.second;
});
```

Q7. Purpose of ‘std::enable_if‘:

‘std::enable_if‘ is used to enable or disable template instantiations based on a condition. Example:

```
template <typename T>
typename std::enable_if<std::is_integral<T>::value, T>::type
increment(T x) { return x + 1; }
```

Q8. Difference Between ‘std::move‘ and ‘std::forward‘:

- ‘std::move‘ casts an object to an rvalue reference, enabling move semantics. - ‘std::forward‘ preserves the value category of the argument.

Example:

```
template <typename T>
void process(T&& arg) {
    T obj = std::forward<T>(arg);
}
```

Q9. Output of the threading code:

Output:

```
Hello from thread!
```

Explanation: The ‘std::thread‘ is detached, so its execution is independent of the main thread. However, the output might not appear if the main thread exits before the detached thread runs.

Q10. Thread Safety with ‘std::atomic‘:

‘std::atomic‘ ensures thread safety by providing atomic operations. Example:

```
#include <atomic>
std::atomic<int> counter(0);
void increment() {
    ++counter;
}
```

Q11. Differences Between ‘virtual‘, ‘override‘, and ‘final‘:

- ‘virtual‘: Marks a member function as polymorphic. - ‘override‘: Ensures a function overrides a base class function. - ‘final‘: Prevents further overrides.

Example:

```
class Base {
    virtual void foo() = 0;
};
class Derived : public Base {
    void foo() override final;
};
```

Q12. Perfect Forwarding:

Perfect forwarding preserves the value category of arguments:

```
template <typename T>
void forwarder(T&& arg) {
    process(std::forward<T>(arg));
}
```

Q13. Iterator Invalidation in ‘std::string‘:

Resizing a ‘std::string‘ invalidates iterators because memory may be reallocated. Example:

```
std::string str = "hello";
auto it = str.begin();
str.resize(10);
// ‘it‘ is now invalid.
```

Q14. ‘std::map‘ vs. ‘std::unordered_map‘:

- ‘std::map‘ is ordered, and operations are $O(\log n)$. - ‘std::unordered_map‘ is unordered, and operations are average $O(1)$.

Example:

```
std::map<int, int> ordered;
std::unordered_map<int, int> unordered;
```