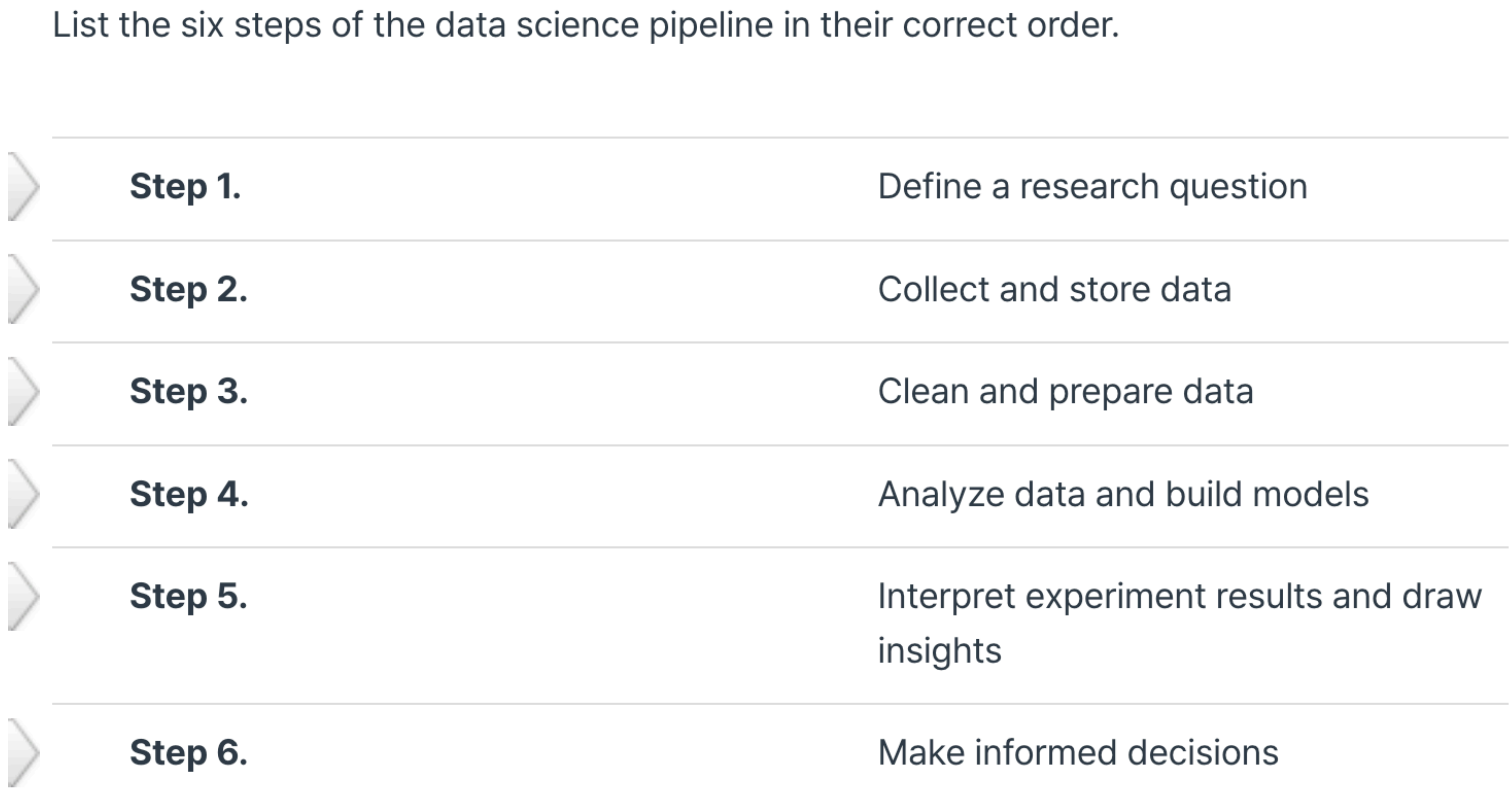
Quiz 1

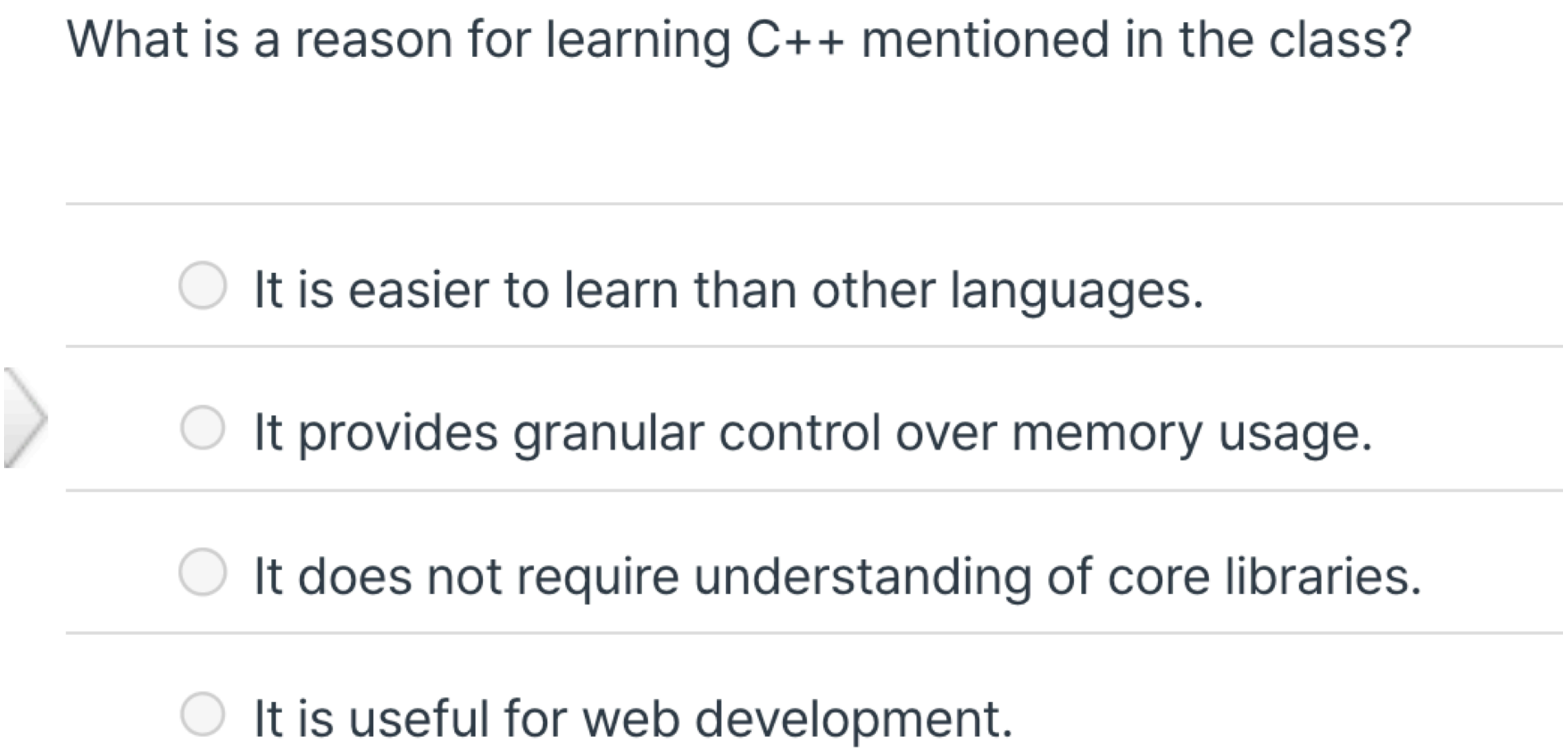
Quiz 1



List the six steps of the data science pipeline in their correct order.

Step 1.	Define a research question
Step 2.	Collect and store data
Step 3.	Clean and prepare data
Step 4.	Analyze data and build models
Step 5.	Interpret experiment results and draw insights
Step 6.	Make informed decisions

Quiz 1



What is a reason for learning C++ mentioned in the class?

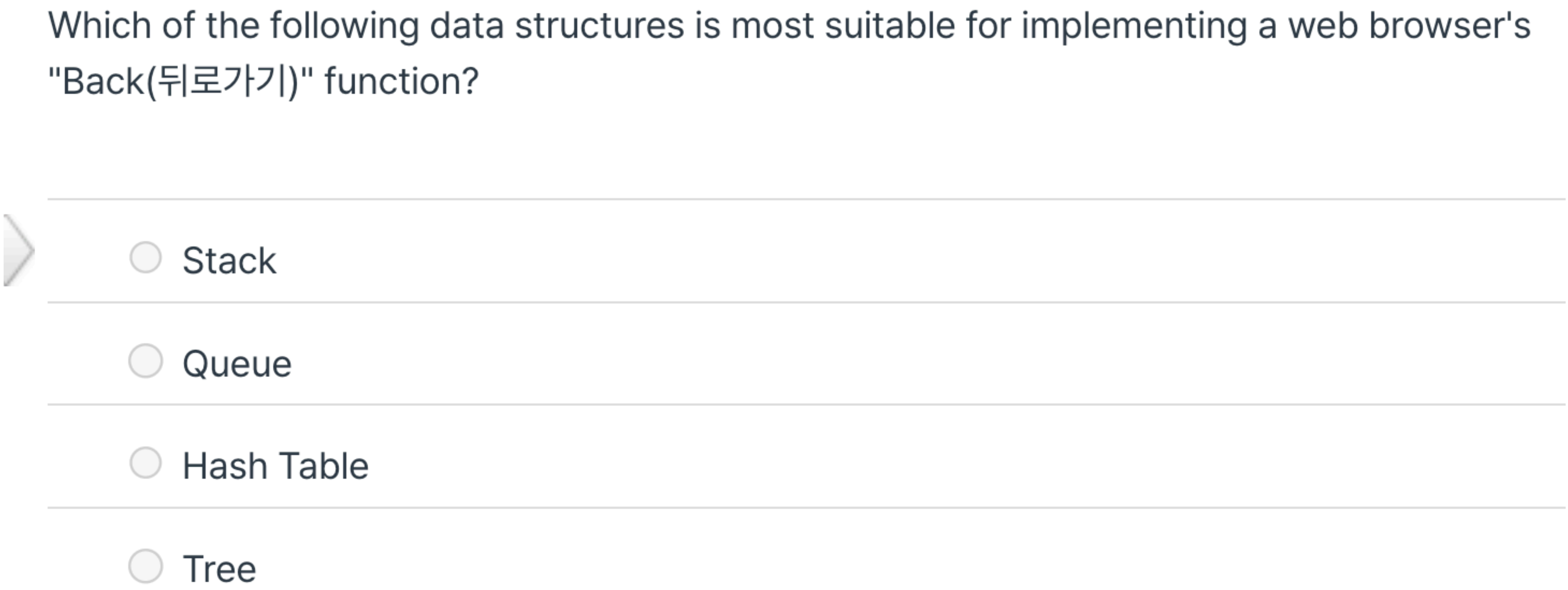
○ It is easier to learn than other languages.

○ It provides granular control over memory usage.

○ It does not require understanding of core libraries.

○ It is useful for web development.

Quiz 1



Which of the following data structures is most suitable for implementing a web browser's "Back(뒤로가기)" function?

○ Stack

○ Queue

○ Hash Table

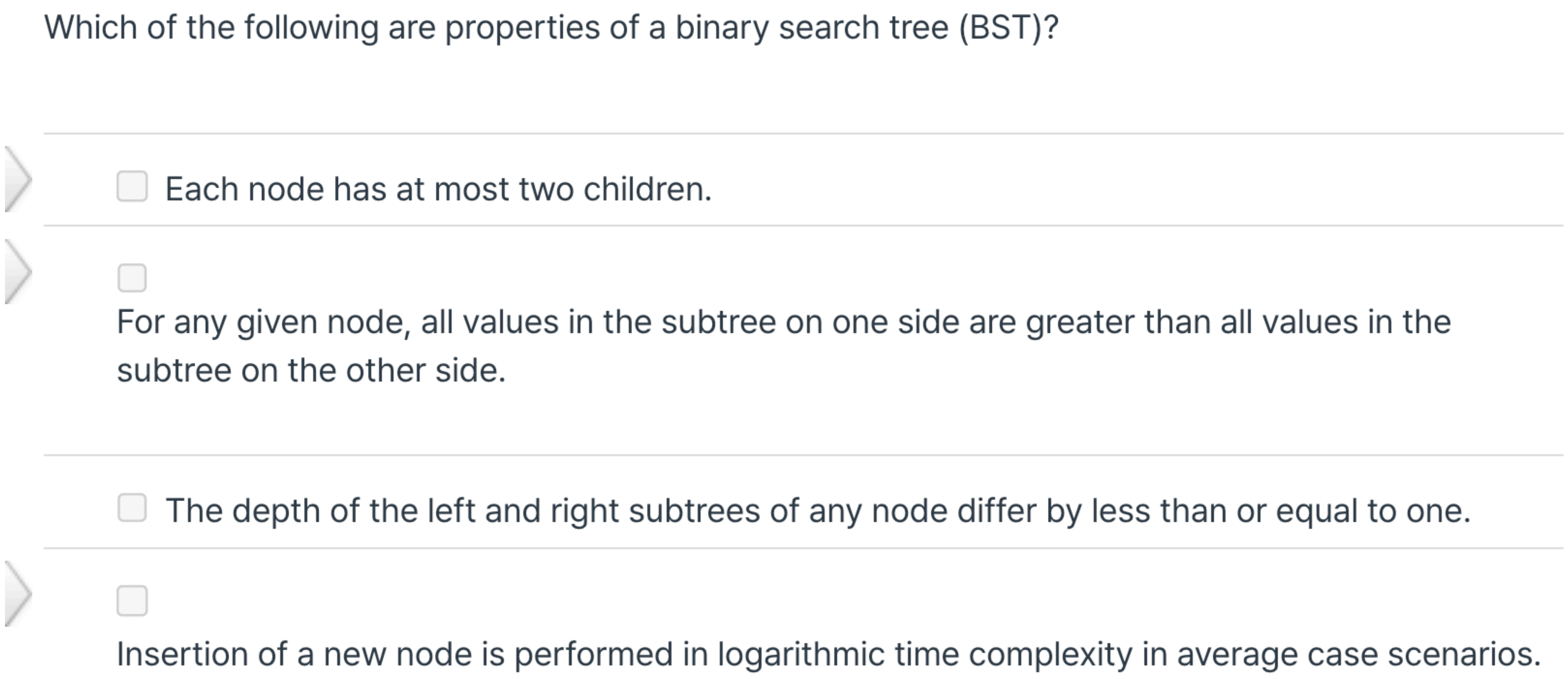
○ Tree

Quiz 1

Which Object-Oriented Programming (OOP) principle is characterized by exposing only high-level interfaces to the outside world, hiding implementation details, and simplifying the interface?

Abstraction

Quiz 1



Which of the following are properties of a binary search tree (BST)?

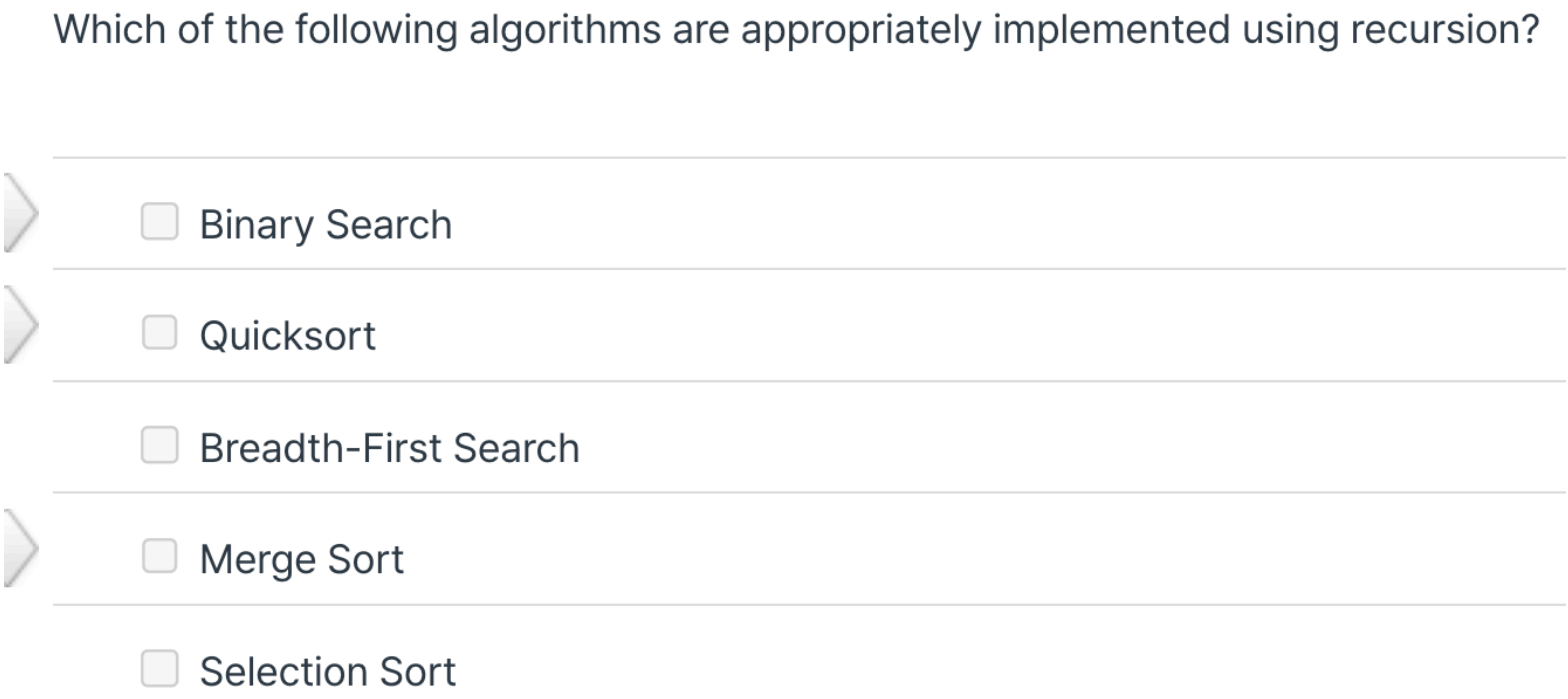
□ Each node has at most two children.

□ For any given node, all values in the subtree on one side are greater than all values in the subtree on the other side.

□ The depth of the left and right subtrees of any node differ by less than or equal to one.

□ Insertion of a new node is performed in logarithmic time complexity in average case scenarios.

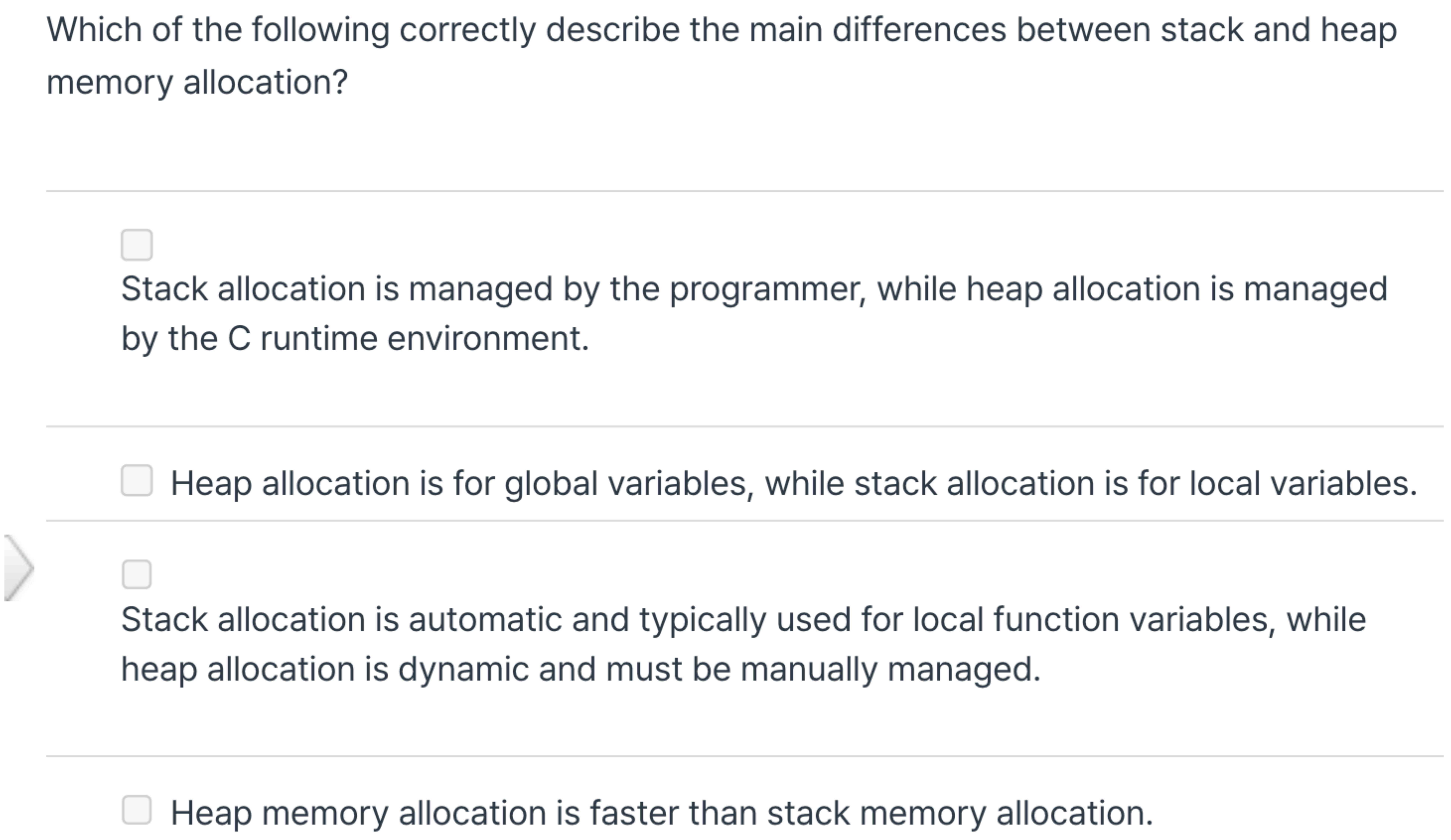
Quiz 1



Which of the following algorithms are appropriately implemented using recursion?

- ❯ □ Binary Search
- ❯ □ Quicksort
- □ Breadth-First Search
- ❯ □ Merge Sort
- □ Selection Sort

Quiz 1



Which of the following correctly describe the main differences between stack and heap memory allocation?

□ Stack allocation is managed by the programmer, while heap allocation is managed by the C runtime environment.

□ Heap allocation is for global variables, while stack allocation is for local variables.

□ Stack allocation is automatic and typically used for local function variables, while heap allocation is dynamic and must be manually managed.

□ Heap memory allocation is faster than stack memory allocation.

Quiz 1

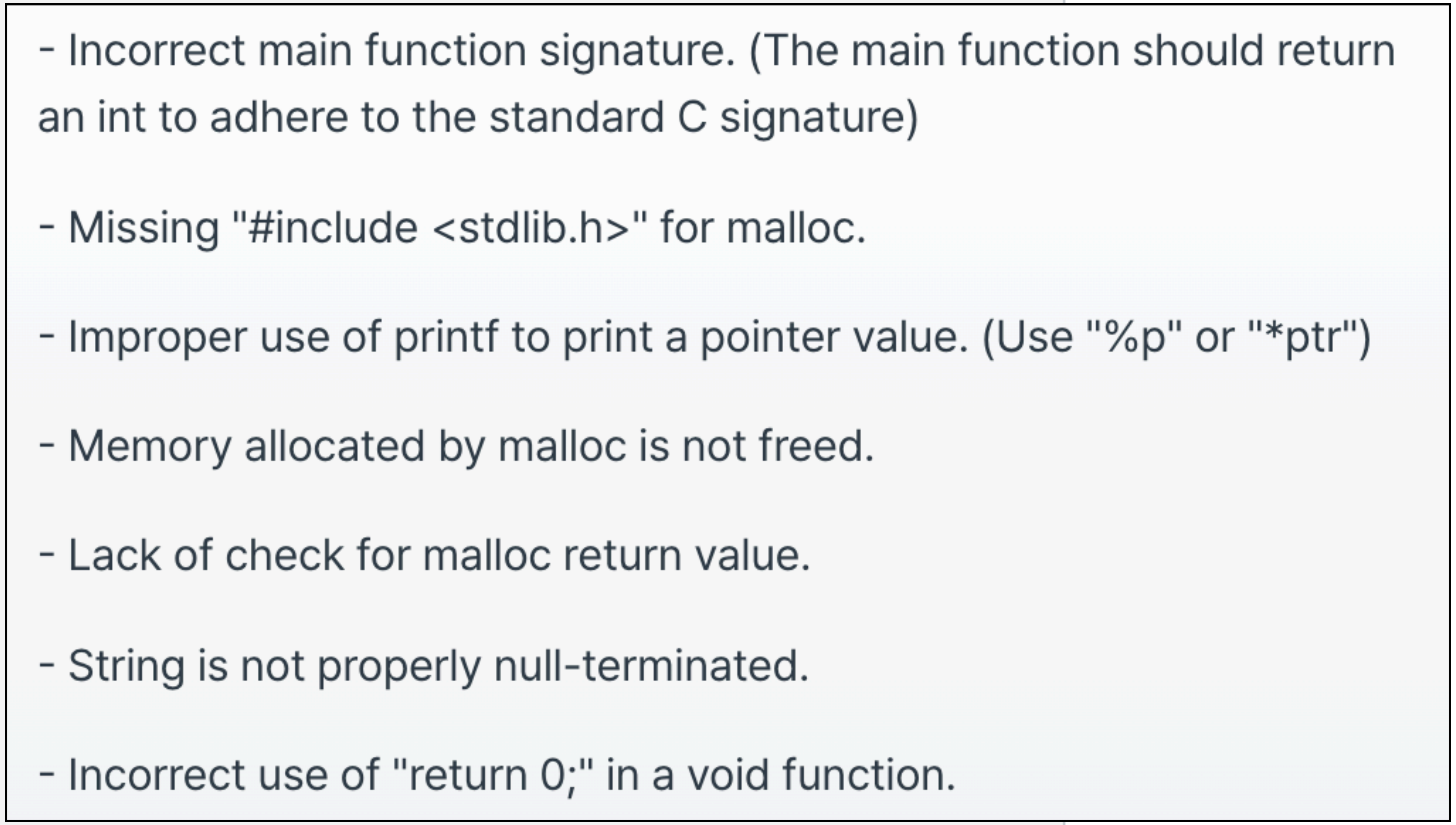
Analyze the C code below and identify all mistakes or bad practices.

```c
#include <stdio.h>

void main() {
    int var = 20;
    int *ptr = &var;
    printf("%d", ptr);

    char *str = malloc(10);
    str[0] = 'a';
    str[1] = 'b';
    str[2] = 'c';
    printf("%s", str);

    return 0;
}
```



- Incorrect main function signature. (The main function should return an int to adhere to the standard C signature)

- Missing "#include <stdlib.h>" for malloc.

- Improper use of printf to print a pointer value. (Use "%p" or "*ptr")

- Memory allocated by malloc is not freed.

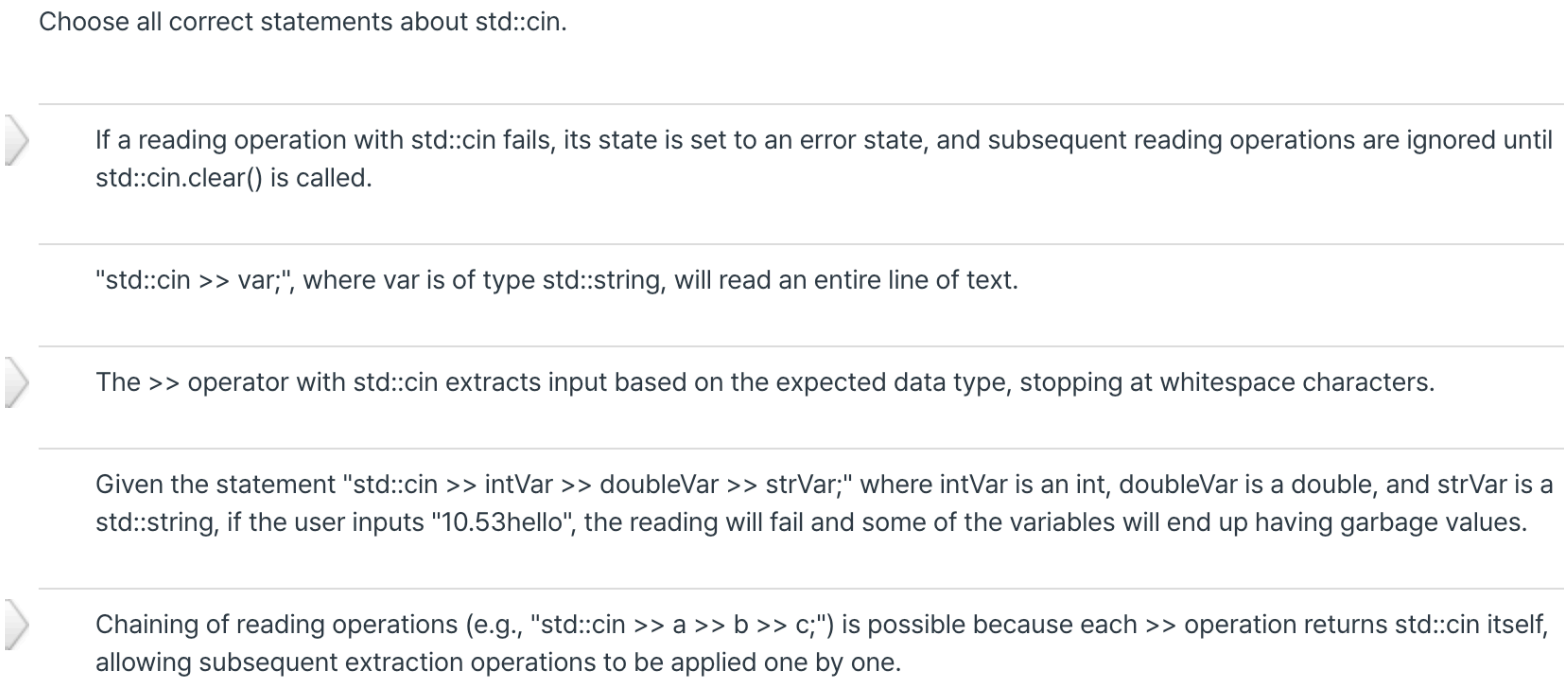
- Lack of check for malloc return value.

- String is not properly null-terminated.

- Incorrect use of "return 0;" in a void function.

Quiz 2

Quiz 2



Choose all correct statements about std::cin.

▷ If a reading operation with std::cin fails, its state is set to an error state, and subsequent reading operations are ignored until std::cin.clear() is called.

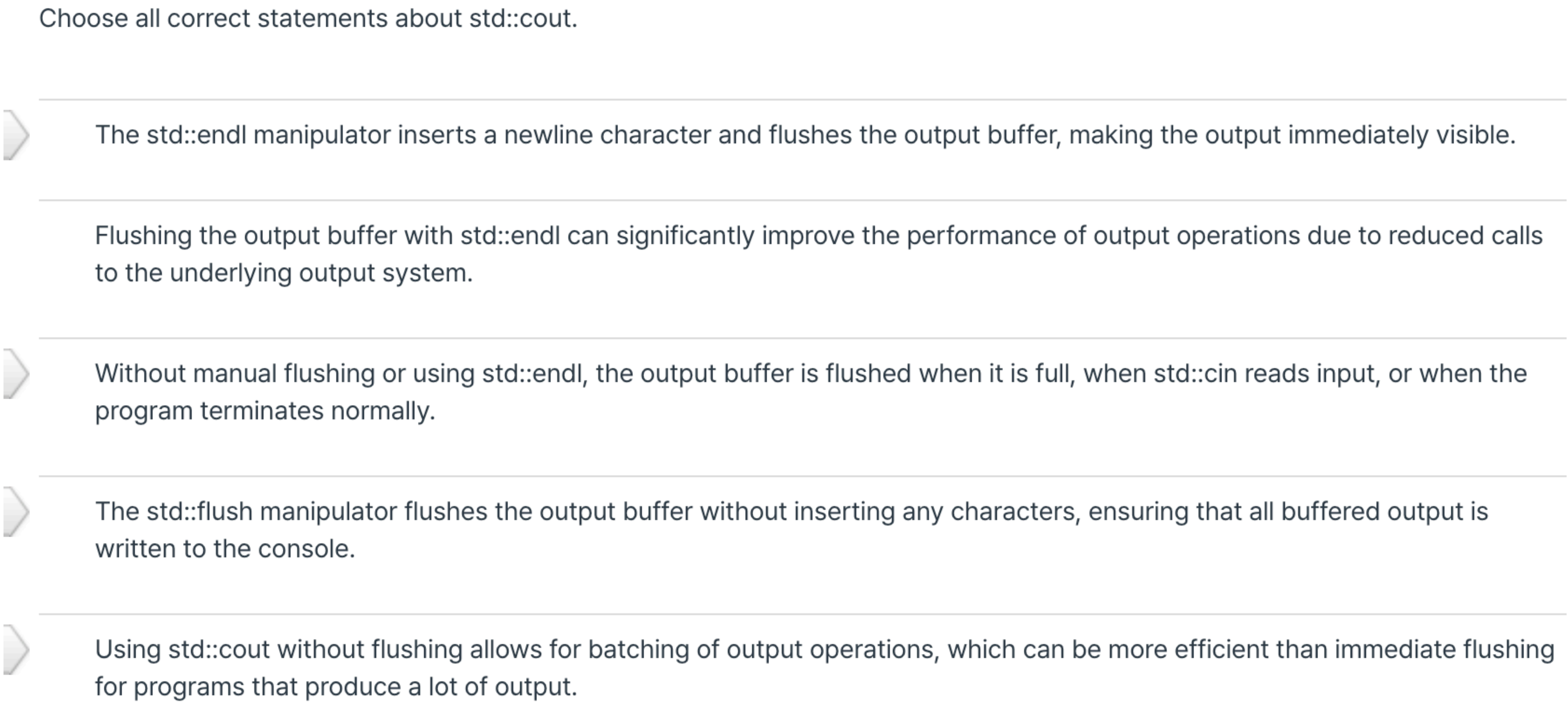
"std::cin >> var;", where var is of type std::string, will read an entire line of text.

▷ The >> operator with std::cin extracts input based on the expected data type, stopping at whitespace characters.

Given the statement "std::cin >> intVar >> doubleVar >> strVar;" where intVar is an int, doubleVar is a double, and strVar is a std::string, if the user inputs "10.53hello", the reading will fail and some of the variables will end up having garbage values.

▷ Chaining of reading operations (e.g., "std::cin >> a >> b >> c;") is possible because each >> operation returns std::cin itself, allowing subsequent extraction operations to be applied one by one.

Quiz 2



Choose all correct statements about std::cout.

❯ The std::endl manipulator inserts a newline character and flushes the output buffer, making the output immediately visible.

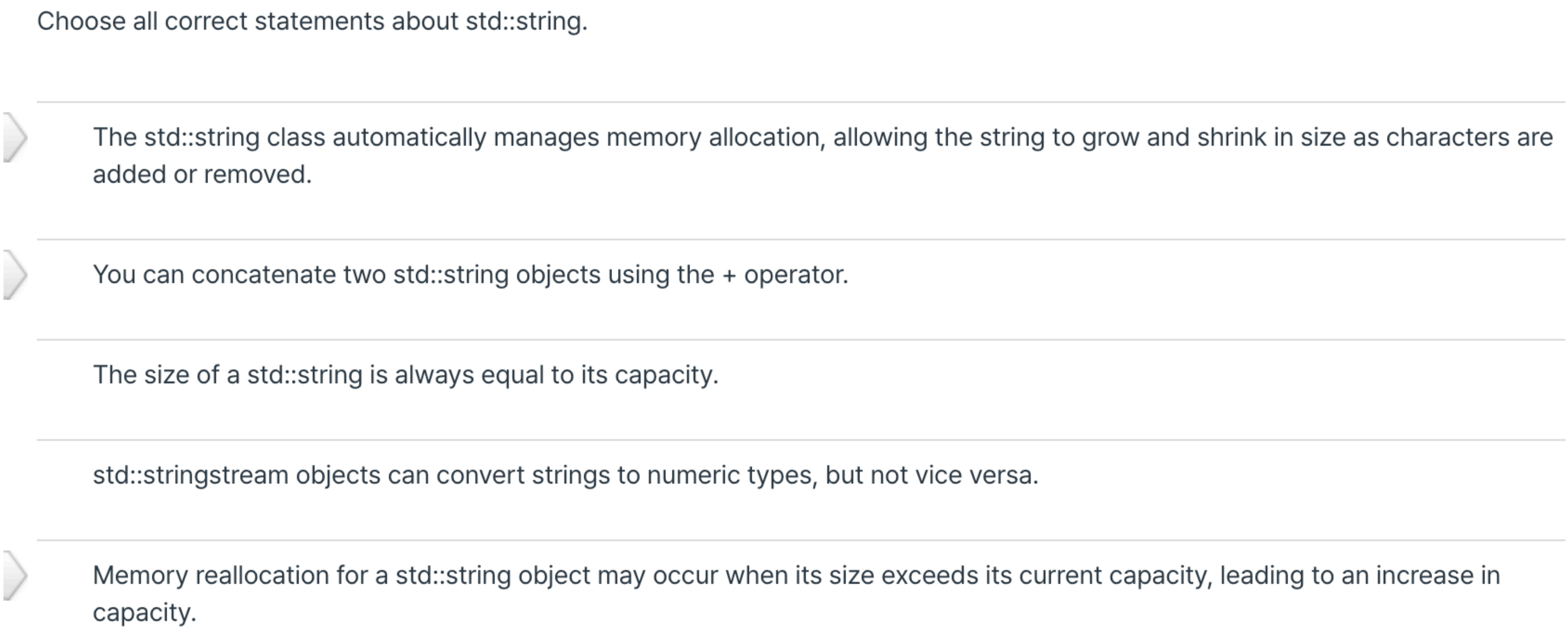
Flushing the output buffer with std::endl can significantly improve the performance of output operations due to reduced calls to the underlying output system.

❯ Without manual flushing or using std::endl, the output buffer is flushed when it is full, when std::cin reads input, or when the program terminates normally.

❯ The std::flush manipulator flushes the output buffer without inserting any characters, ensuring that all buffered output is written to the console.

❯ Using std::cout without flushing allows for batching of output operations, which can be more efficient than immediate flushing for programs that produce a lot of output.

Quiz 2



Choose all correct statements about std::string.

▷ The std::string class automatically manages memory allocation, allowing the string to grow and shrink in size as characters are added or removed.

▷ You can concatenate two std::string objects using the + operator.

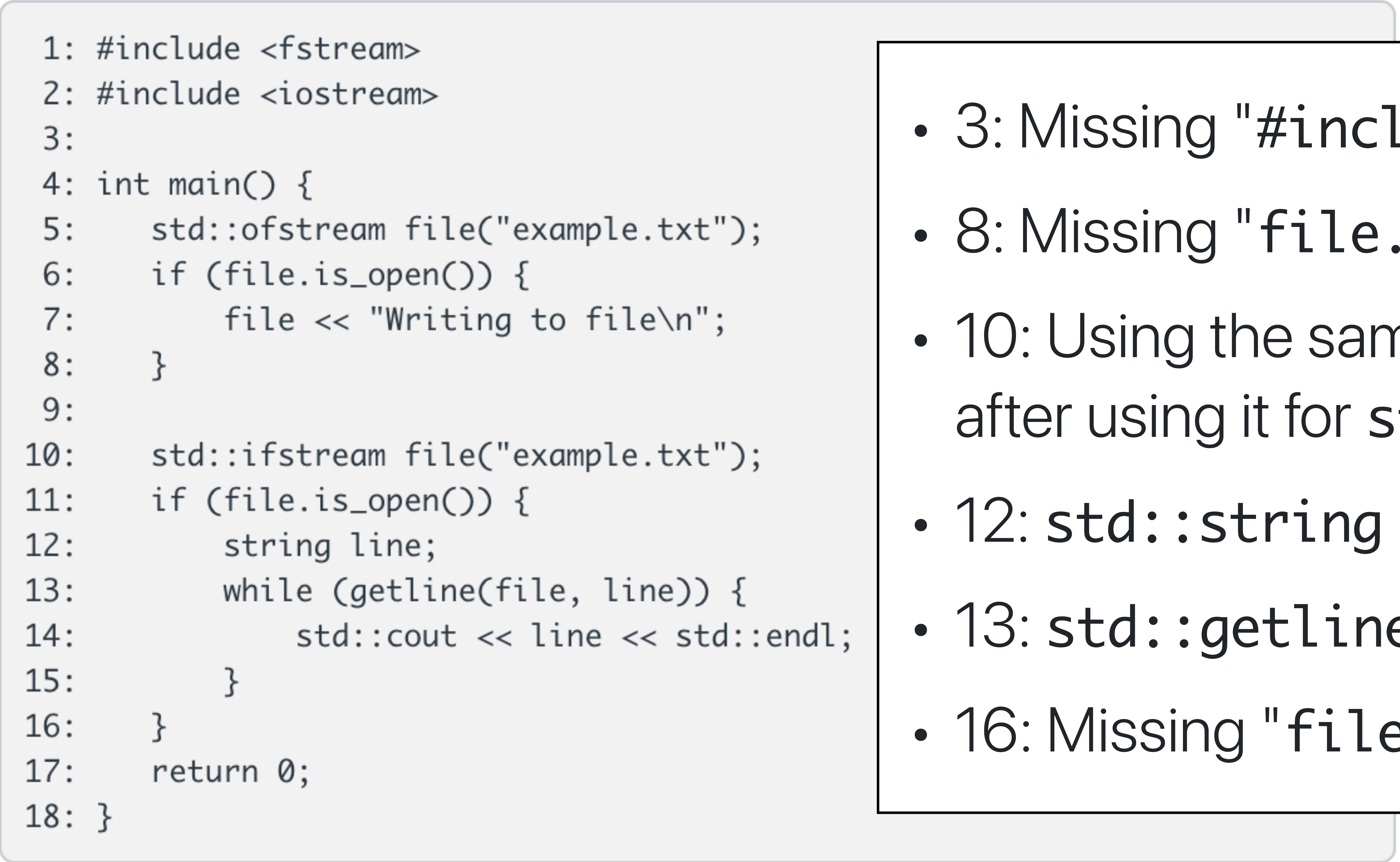
The size of a std::string is always equal to its capacity.

std::stringstream objects can convert strings to numeric types, but not vice versa.

▷ Memory reallocation for a std::string object may occur when its size exceeds its current capacity, leading to an increase in capacity.

Quiz 2

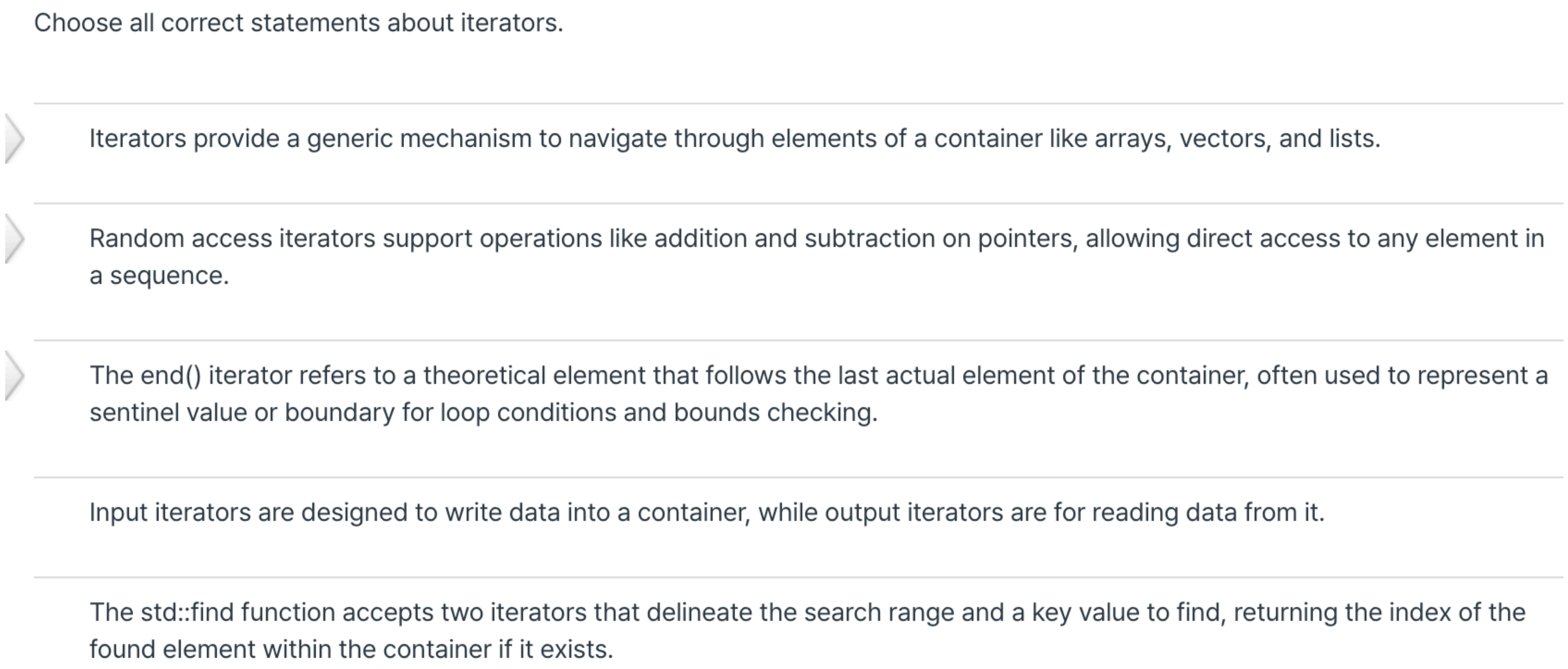
Read the following C++ code snippet related to file I/O. Identify all line numbers that contain mistakes or demonstrate bad practices. If any essential statements are missing, specify the line numbers where those statements would be most appropriately added.



```
 1: #include <fstream>
 2: #include <iostream>
 3:
 4: int main() {
 5:     std::ofstream file("example.txt");
 6:     if (file.is_open()) {
 7:         file << "Writing to file\n";
 8:     }
 9:
10:     std::ifstream file("example.txt");
11:     if (file.is_open()) {
12:         string line;
13:         while (getline(file, line)) {
14:             std::cout << line << std::endl;
15:         }
16:     }
17:     return 0;
18: }
```

- 3: Missing "`#include <string>`"
- 8: Missing "`file.close();`"
- 10: Using the same variable for `std::ifstream` after using it for `std::ofstream`
- 12: `std::string`
- 13: `std::getline`
- 16: Missing "`file.close();`"

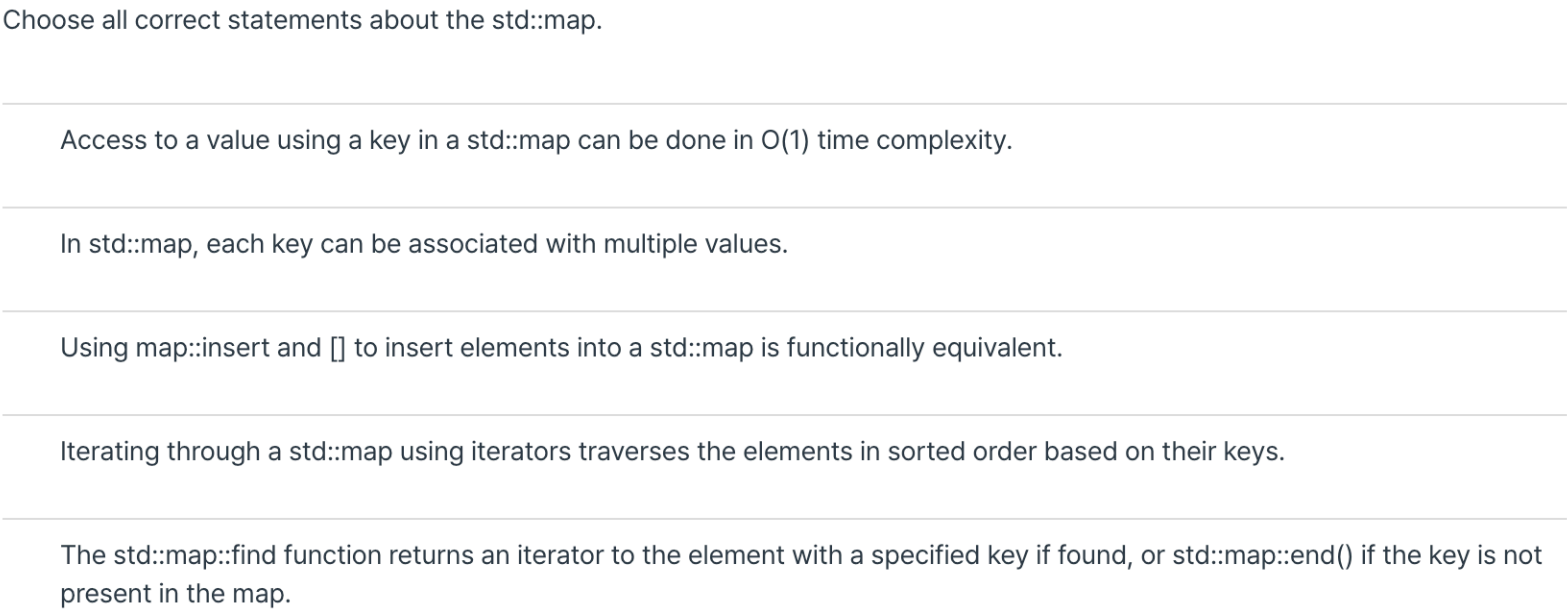
Quiz 2



Choose all correct statements about iterators.

- ❯ Iterators provide a generic mechanism to navigate through elements of a container like arrays, vectors, and lists.
- ❯ Random access iterators support operations like addition and subtraction on pointers, allowing direct access to any element in a sequence.
- ❯ The end() iterator refers to a theoretical element that follows the last actual element of the container, often used to represent a sentinel value or boundary for loop conditions and bounds checking.
- Input iterators are designed to write data into a container, while output iterators are for reading data from it.
- The std::find function accepts two iterators that delineate the search range and a key value to find, returning the index of the found element within the container if it exists.

Quiz 2



Choose all correct statements about the std::map.

Access to a value using a key in a std::map can be done in O(1) time complexity.

In std::map, each key can be associated with multiple values.

Using map::insert and [] to insert elements into a std::map is functionally equivalent.

❯ Iterating through a std::map using iterators traverses the elements in sorted order based on their keys.

❯ The std::map::find function returns an iterator to the element with a specified key if found, or std::map::end() if the key is not present in the map.

Quiz 3

Class Templates

The following code defines a class template Tuple that stores two elements. However, there are errors in the code. Identify the lines that contain the error(s).

```
 1. template <typename T1, typename T2>
 2. class Tuple {
 3. private:
 4.     T1 first;
 5.     T2 second;
 6.
 7. public:
 8.     Tuple(T1 f, T2 s) : first(f), second(s) {}
 9.
10.     void print() const {
11.         std::cout << first << ", " << second << std::endl;
12.     }
13.
14.
✓15.     void setTuple(T1 newFirst, T1 newSecond) {
16.         first = newFirst;
✓17.         second = newSecond;
18.     }
19.
20.
21.     void swap() {
22.         T1 temp = first;
✓23.         first = second;
✓24.         second = temp;
25.     }
26. };
```

Operator Overloading

Match the following operators to their most appropriate return type when overloaded in a custom container class like SimpleVector.

operator++() (Post Increment)	SimpleVector<T>
operator+= (Addition Assignment)	SimpleVector<T>&
operator[] (Subscript Operator)	T&
operator== (Equality Check)	bool

Operator Overloading

When overloading the postfix increment operator (operator++(int)) for a SimpleVector class, you need to return the vector before it was incremented. To do this, you typically store a temporary copy of the object. Fill in the blank in the following code to correctly implement this operator.

```
template <typename T>
SimpleVector<T> SimpleVector<T>::operator++(int) {
    SimpleVector<T> temp = _*this_;  // Store current state
    for (int i = 0; i < size; ++i) {
        array[i]++;
    }
    return temp;
}
```

Operator Overloading

Consider the following overloaded equality operator (operator==) for a SimpleVector class. What would be the primary concern if the input argument `rhs` is not declared `const`?

```
template<typename T>
bool SimpleVector<T>::operator==(SimpleVector<T>& rhs) {
    if (size != rhs.getSize()) return false;
    for (int i = 0; i < size; ++i) {
        if (array[i] != rhs[i]) return false;
    }
    return true;
}
```

Equality cannot be checked if the operand is a const SimpleVector variable

Copy and Move Semantics

Below are four different code snippets involving object creation and assignment for a class ResourceHandler. Match each snippet with one of the following:

- Copy Constructor
- Copy Assignment Operator
- Move Constructor
- Move Assignment Operator

Copy and Move Semantics

Code 1:

```
ResourceHandler rh1;
ResourceHandler rh2;
rh1 = rh2;
```

Copy Assignment Operator

Copy and Move Semantics

Code 2:

```
ResourceHandler createHandler() {
    ResourceHandler temp;
    return temp;
}
ResourceHandler rh1;
rh1 = createHandler();
```

Move Assignment Operator

Copy and Move Semantics

Code 3:

```
ResourceHandler createHandler() {
    return ResourceHandler();
}
ResourceHandler rh1 = createHandler();
```

Move Constructor

Copy and Move Semantics

Code 4:

```
ResourceHandler rh1;
ResourceHandler rh2 = rh1;
```

Copy Constructor