

Interview5

Stage 1: Introduction,critical thinking (7/10)

Stage2: Oops (4/10)

Problem area: *(No any clear explanation so marks deducted)*

1. Which of the following type of class allows only one object of it to be created?

Virtual class

Abstract class

Singleton class(0.5)

Friend class

Solution: C is correct

A singleton class is a special class in C++ that can only have one instance at a time. This is achieved by making the constructor private and providing a static method that returns the instance of the class.

Here is an example of a singleton class in C++:

```
C++
class Singleton {
private:
    static Singleton* instance;
    Singleton() {}
public:
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }
};

Singleton* Singleton::instance = nullptr;

int main() {
    Singleton* s1 = Singleton::getInstance();
    Singleton* s2 = Singleton::getInstance();

    // s1 and s2 will point to the same object
    return 0;
}
```

2.

Which of the following is not a type of constructor?

Copy constructor

Friend constructor

Default constructor

Parameterized constructor

Solution: reference link <https://byjusexamprep.com/gate-cse/which-of-the-following-is-not-a-type-of-constructor#:~:text=Friend%20Constructor%20is%20not%20a%20type%20of%20constructor%20in%20C%2B%2B,-Solution&text=The%20answer%20to%20the%20question,Default%20Constructors>

3.

Which of the following statements is correct?

Base class pointer cannot point to derived class.

Derived class pointer cannot point to base class.

Pointer to derived class cannot be created.

Pointer to base class cannot be created.

Solution: B

In C++ programming, it is possible for a base class pointer to point to an object of a derived class, but a derived class pointer cannot point to an object of a base class.

Here's a detailed explanation and an example:

Base Class Pointer to Derived Class Object:

In C++, a base class pointer can point to an object of a derived class. This is a form of polymorphism, where a base class reference or pointer can refer to objects of its derived classes.

This allows you to write generic code that works with a base class but can handle objects of derived classes as well.

```
class Base {
public:
    virtual void display() {
        std::cout << "Base display" << std::endl;
    }
}
```

```
};

class Derived : public Base {
public:
    void display() override {
        std::cout << "Derived display" << std::endl;
    }
};
```

```
int main() {
    Base* basePtr = new Derived(); // Base pointer pointing to a derived class object
    basePtr->display(); // Calls the overridden method in the derived class
    delete basePtr;
    return 0;
}
```

In the above example, basePtr is a pointer to a base class (Base), but it is pointing to an object of a derived class (Derived). This is allowed in C++.

Derived Class Pointer to Base Class Object:

However, the reverse is not true in C++. A derived class pointer cannot point to an object of a base class.

This is because the derived class might have additional members and methods not present in the base class. Pointing to an object of the base class would mean the derived class pointer would be pointing to an object that doesn't have those additional members or methods, which could lead to errors.

For example:

```
class Base {
public:
    void baseFunction() {
        std::cout << "Base function" << std::endl;
    }
};
```

```
class Derived : public Base {
public:
    void derivedFunction() {
```

```
        std::cout << "Derived function" << std::endl;
    }
};

int main() {
    Base* baseObj = new Base();

    // The following line would cause a compilation error:

    // Derived* derivedPtr = baseObj; // Invalid: Cannot point derived class pointer to base class object

    delete baseObj;

    return 0;
}
```

In the above code snippet, the line `Derived* derivedPtr = baseObj;` is not allowed because it attempts to assign a base class object to a derived class pointer, which is not valid. This assignment would not make sense because `derivedPtr` would expect additional members or methods that are not present in `baseObj`.

4.

Which of the following is not the member of class?

Static function

Friend function

Const function

Virtual function

solution: friend function is not the member function of class B

5.

Which of the following concepts means determining at runtime what method to invoke?

Data hiding

Dynamic Typing

Dynamic binding

Dynamic loading

Solution: The correct answer to the question "Which of the following concepts means determining at runtime what method to invoke?" is:

c. Dynamic binding

Dynamic binding (also known as late binding) is a concept in object-oriented programming where the method to invoke is determined at runtime, based on the actual object type that the pointer or reference is pointing to. This allows for polymorphism, where a base class pointer or reference can refer to an object of a derived class, and the appropriate method in the derived class is invoked.

. Data hiding:

- Data hiding is a principle of encapsulation in object-oriented programming where the internal state of an object is hidden from the outside world. This is achieved through access modifiers such as `private` and `protected` in a class, restricting access to the class's data members.
- It is not directly related to determining which method to invoke at runtime.

b. Dynamic Typing:

- Dynamic typing is a characteristic of programming languages where types are determined at runtime rather than compile-time. This means that variables are assigned types based on the data they hold, and their types can change during the execution of a program.
- While dynamic typing might involve runtime decisions about types, it does not directly relate to determining which method to invoke.

d. Dynamic Loading:

- Dynamic loading, also known as dynamic linking, refers to the ability of a program to load libraries or modules at runtime rather than at compile-time. This allows for more flexible and modular program design, as different components can be loaded and unloaded as needed.
- It is not directly related to method invocation or determining which method to invoke at runtime.

The correct option from the provided choices is **c. Dynamic binding**, which directly pertains to determining at runtime which method to invoke based on the actual type of the object being referenced.

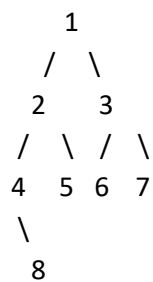
Stage 3: DSA Round(Total marks 20+30=50)

Question 1: (Marks-> 15/20)

Problem: Intuition was correct but very complex and inefficient approach, But overall after my help able to solve that's a plus point

Given a Binary Tree, return Left view of it. Left view of a Binary Tree is set of nodes visible when tree is visited from Left side. The task is to complete the function leftView(), which accepts root of the tree as argument. If no left view is possible, return an empty tree.

Left view of following tree is 1 2 4 8.



Example 1:

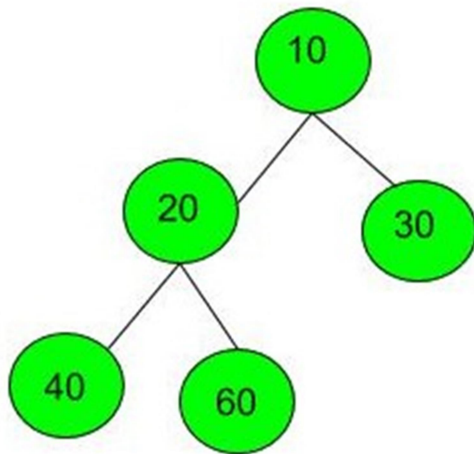
Input:



Output: 1 3

Example 2:

Input:



Output: 10 20 40

Your Task:

You just have to complete the function `leftView()` that returns an array containing the nodes that are in the left view. The newline is automatically appended by the driver code.

Expected Time Complexity: $O(N)$.

Expected Auxiliary Space: $O(N)$.

Constraints:

$0 \leq \text{Number of nodes} \leq 100$

$0 \leq \text{Data of a node} \leq 1000$

Reference: <https://www.geeksforgeeks.org/problems/left-view-of-binary-tree/1>

```
void solve(vector<int>&ans, Node*root, int level)
{
    if(!root)
        return;

    if(ans.size() < level)
        ans.push_back(root->data);

    solve(ans, root->left, level+1);
    solve(ans, root->right, level+1);
}

vector<int> leftView(Node *root)
{
    vector<int> ans;
    solve(root, ans, 1);  →wrong input parameter inputs(ans,root,1)
    return ans;
}
```

```
}
```

Question 2:-> (10/30)

Problem: could not able to convert logic to code,

Given an integer array `coins[]` of size `N` representing different denominations of currency and an integer `sum`, find the number of ways you can make `sum` by using different combinations from `coins[]`.

Note: Assume that you have an infinite supply of each type of coin. And you can use any coin as many times as you want.

Example 1:

Input:

`N = 3, sum = 4`

`coins = {1,2,3}`

Output: 4

Explanation: Four Possible ways are: `{1,1,1,1}`, `{1,1,2}`, `{2,2}`, `{1,3}`.

Example 2:

Input:

`N = 4, Sum = 10`

`coins = {2,5,3,6}`

Output: 5

Explanation: Five Possible ways are: `{2,2,2,2,2}`, `{2,2,3,3}`, `{2,2,6}`, `{2,3,5}` and `{5,5}`.

Your Task:

You don't need to read input or print anything. Your task is to complete the function `count()` which accepts an array `coins` its size `N` and `sum` as input parameters and returns the number of ways to make change for given sum of money.

Expected Time Complexity: $O(\text{sum} * N)$

Expected Auxiliary Space: $O(\text{sum})$





Constraints:

$1 \leq \text{sum}, N, \text{coins}[i] \leq 103$

Reference : <https://www.geeksforgeeks.org/problems/coin-change2448/1>

React coding problem (marks 30/40)

Problem: could not able to implement scroll section so CSS is weak point,
lets assume you have given a backend api for fetching chats (api: <http://xyz.com/chat>) this api will
response with array of objects ({id,sender,receiver,message})
store these chat in context api and create a chat scroll section with sender name and chat as
description.

	Gavnish kumar <i>Hey subrat how are you</i>
	jaanu <i>Jaanu tera mr gya</i>
	Jaaneman <i>Ja mr ja tera mera break up.</i>
	babu sona <i>I love you sona</i>

Write a javascript function that determine sum of top 5 maximum numbers in array. (marks 9/10)
(10min)

React theory question.

Q-1 Can you use multiple contexts in a single application? (4/5)

The interviewer wants to know if you can use multiple contexts in a React application.

How to answer: Yes, you can use multiple contexts in a single application by creating separate context objects and using them as needed in different parts of your app.

Example Answer: "Yes, you can use multiple contexts in a single application. You can create separate context objects for different types of data you want to share and use them as needed in different parts of your app. This allows for a clean separation of concerns and prevents data from being mixed between unrelated components."

When should you use React Context API over props drilling? (3/5)

The interviewer is interested in your understanding of when to choose React Context API over passing props down the component tree.

How to answer: Explain that React Context API is a better choice when you have to pass data through many levels of components, making it more efficient and maintaining a cleaner code structure.

Example Answer: *"You should use React Context API over props drilling when you need to pass data through many levels of components. Props drilling can become cumbersome and error-prone when your component tree is deep. Context API simplifies this process, improves code maintainability, and reduces the likelihood of errors."*

Can you update the context data using the Provider component? (0/5)

The interviewer wants to know if you can update the context data once it's been provided using the Provider component.

How to answer: Explain that you can update the context data by changing the `value` prop of the `Provider` component. However, this will trigger a re-render of all components consuming that context.

Example Answer: *"Yes, you can update the context data by changing the `value` prop of the `Provider` component. However, it's important to note that updating the context data will trigger a re-render of all components that consume that context. So, you should use this feature carefully to avoid unnecessary re-renders."*

How can you optimize performance when using React Context API? (1/5)

The interviewer wants to know about performance considerations when using React Context API.

How to answer: Mention techniques like memoization, using the `useMemo` hook, and avoiding excessive context usage to optimize performance.

Example Answer: *"To optimize performance with React Context API, you can use memoization techniques to prevent unnecessary re-renders. The `useMemo` hook is helpful for memoizing values. Additionally, avoid excessive context usage and consider breaking up your context into smaller, more focused providers to minimize the impact of context changes on your components."*

How can you handle async operations in React Context? (3/5)

The interviewer is interested in how you handle asynchronous operations within the context of a React application.

How to answer: Explain that you can use techniques like `useEffect` and `async/await` for handling async operations within a component that consumes context data.

Example Answer: *"To handle async operations in React Context, you can use the `useEffect` hook in a component that consumes context data. You can perform async operations within the `useEffect` function, making use of `async/await` to ensure your code runs smoothly while fetching data or making asynchronous calls."*

How can you test components that use React Context API? (1/5)

The interviewer is interested in your knowledge of testing components that rely on React Context.

How to answer: Explain that you can test components using React Context by providing a test context using `TestRenderer` or a testing library like `react-testing-library` and `Enzyme`.

Example Answer: *"To test components using React Context, you can provide a test context with predefined values using testing libraries like `react-testing-library` or `Enzyme`. This allows you to simulate context data and test how components interact with it without relying on the actual application context."*

Marks Obtained: 87

Total marks: 150