useCallback Hook

- The useCallback hook is a React hook used to memoize a callback function.
- It ensures that a function instance is not re-created on every render unless its dependencies change.
- This optimization helps prevent unnecessary re-renders and improves the performance of React applications, especially when passing callbacks to child components or using them in event handlers.
- React's useCallback hook is designed to memoize a function to ensure that the same function reference is preserved across renders unless its dependencies change.
- By memoizing a function, you can optimize performance by avoiding the creation of a new function instance on every render.

Key Concepts

- **Memoization:** Storing the result of a computation or function so it can be reused without recalculating.
- **Preventing Function Recreation:** Functions in JavaScript are objects, and each render creates a new function instance. useCallback avoids this by returning the same function reference unless dependencies change.

Problem Without useCallback

In React, functions are treated as objects. During every render, JavaScript creates new instances of the functions defined inside a component. Even if the logic inside the function remains unchanged, the function's reference changes.

This behavior can lead to:

- **Unnecessary Re-renders:** When a function is passed as a prop to a child component, a new function reference forces the child component to re-render, even if nothing meaningful has changed.
- **Wasted Computation:** If the function performs a heavy computation, recreating it on every render becomes expensive.

How useCallback Solves the problem

The useCallback hook solves the above problem by returning the same function reference as long as the dependencies remain unchanged. It memoizes the function, ensuring stable references for components or hooks that depend on it.

Key Benefits

- Prevents unnecessary re-renders of child components when the function is passed as a prop.
- Reduces the computational cost of recreating the function.

Syntax:

```
const memoizedCallback = useCallback(callbackFunction, [dependencies]);
```

where,

- callbackFunction: The function you want to memoize.
- dependencies: An array of values that useCallback depends on. The memoized function will only update when one of these values changes.

Example

```
import React, { useState, useCallback } from 'react';
function App() {
  const [count, setCount] = useState(0);

  // Memoize the increment function
  const increment = useCallback(() => {
                              setCount((prevCount) => prevCount + 1);
  }, []); // Dependency array is empty, so the function won't be recreated.
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
export default App;
```

Importance in React Functional Components

Why useCallback Matters:

- **Optimizing Re-Renders:** When a memoized function is passed as a prop to child components, React compares the function's reference to avoid unnecessary renders.
  Without useCallback, the child component may re-render every time the parent renders, as the function reference would change.
- **Improved Performance:** Essential for large applications where components re-render frequently.
  Particularly useful in lists, complex trees, or when working with expensive computations.
- **Interaction with React.memo:** React.memo optimizes functional components by preventing re-renders when props don't change.
  useCallback ensures that functions passed as props are stable, working hand-in-hand with React.memo.

React.memo

- React.memo() is a higher-order component (HOC) in React that optimizes the rendering performance of functional components.
- It prevents unnecessary re-renders by memoizing the component's output, rendering it only when its props change.

How It Works

- React.memo() performs a shallow comparison of the props.
- If the props remain the same between renders, the component is not re-rendered.
- This is similar to PureComponent in class components but used for functional components.

Syntax

```
const MemoizedComponent = React.memo(Component);
```

React.memo : Use Cases

- **Performance Optimization:** When a component renders frequently with the same props but its parent updates often.
- **Static Data:** When the props passed to the component are static or rarely change.
- **Preventing Child Re-Renders:** Useful for child components in a large tree where the parent re-renders frequently.

Example : Without React.memo()

```
import React from 'react';

const Child = ({ name }) => {
  console.log('Child component rendered');
  return <div>Welcome to, {name}!</div>;
};

const App = () => {
  const [count, setCount] = React.useState(0);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <Child name="Learn2Earn Labs" />
    </div>
  );
};

export default App;
```

Behavior: Every time the Increment Count button is clicked, the Child component renders even though its name prop hasn't changed.

Example : With React.memo()

```
import React from 'react';
const Child = React.memo(({ name }) => {
        console.log('Child component rendered');
        return <div>Welcome to, {name}!</div>;
});
const App = () => {
  const [count, setCount] = React.useState(0);
  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <Child name="Learn2Earn Labs" />
</div>
  );};
export default App;
```

Behavior: The Child component only renders once, as its name prop doesn’t change when the button is clicked.

Example: When Props Change

```
import React from 'react';
const Child = React.memo(({ name }) => {
                console.log('Child component rendered');
                return <div>Welcome to, {name}!</div>;
});
const App = () => {
  const [name, setName] = React.useState('Learn2Earn Labs');
  return (
    <div>
      <button onClick={() => setName(name === 'Learn2Earn Labs' ? 'Best Training
Institute' : 'Learn2Earn Labs')}>Change Name</button>
      <Child name={name} />
    </div>
  );};
export default App;
```

Behavior: The Child component renders only when the name prop changes.

Example: Custom Comparison

```
import React from 'react';
const Child = React.memo(
  ({ data }) => {
            console.log('Child component rendered');
            return <div>Data: {data.value}</div>;
  }, //custom comparison
  (prevProps, nextProps) => prevProps.data.value === nextProps.data.value
);
const App= () => {
  const [data, setData] = React.useState({ value: 0 });
  return (
    <div>
      <button onClick={() => setData({ value: data.value })}>Update Data</button>
      <Child data={data} />
    </div>
  );
};
export default App;
```

Behavior: The Child component does not re-render unnecessarily, as the custom comparison ensures data.value is compared.

Custom comparison

(prevProps, nextProps) => prevProps.data.value === nextProps.data.value

is a custom comparison function used in React.memo() to determine whether a component should re-render. This function compares specific parts of the previous and next props (data.value in this case) to decide if the props have actually changed.

- **Default Behavior of React.memo():** By default, React.memo() performs a shallow comparison of props, meaning it compares primitive values (e.g., strings, numbers) but doesn't deeply compare nested objects.
- **Custom Comparison Function:** When props are complex objects (e.g., data is an object), shallow comparison fails because even if the object's properties are the same, the object itself might have a different reference in memory.

The custom comparison function explicitly checks if the property you care about (data.value) has changed, allowing fine-grained control.

Example: Custom Comparison

```
import React from 'react';

const Child = React.memo(
  ({ data }) => {
                    console.log('Child component rendered');
                    return <div>Data: {data.value}</div>;
  },
  (prevProps, nextProps) => prevProps.data.value === nextProps.data.value
);

const App = () => {
  const [data, setData] = React.useState({ value: 0 });

  return (
    <div>
      <button onClick={() => setData({ value: data.value })}>Update Data</button>
      <button    onClick={()    =>    setData({    value:    data.value    +    1
})}>Increment</button>
      <Child data={data} />
    </div>
  );
};

export default App;
```

Points to remember:

- React.memo() prevents re-renders if props haven't changed.
- With custom comparison, you control re-rendering logic.
- Use React.memo() when re-rendering can be avoided to optimize performance.

Example: Optimizing Child Component Rendering using useCallback & memo

```
import React, { useState, useCallback, memo } from 'react';
// Child component that renders only if its props change
const Child = memo(({ onClick }) => {
        console.log('Child component rendered');
        return <button onClick={onClick}>Click Me</button>;
});

function App() {
  const [count, setCount] = useState(0);

  // Memoize the callback to prevent unnecessary child renders
  const handleClick = useCallback(() => {
                                    setCount((prev) => prev + 1);
  }, []);

  return (
    <div>
      <p>Count Value : {count}</p>
      <Child onClick={handleClick} />       // passing callback as prop
    </div>
  );
}

export default App;
```

Handling complex dependencies

- Handling complex dependencies in React hooks like useEffect, useCallback, and useMemo is crucial to ensure efficient rendering and avoid unnecessary side effects.
- When working with hooks, you often need to specify dependencies in their dependency arrays (i.e., the second argument passed to hooks). Improperly managing dependencies can lead to performance issues, unexpected behavior, or redundant executions of side effects.

Using Stable References in the Dependency Array

Problem: If you include non-stable references (e.g., objects or functions) in the dependency array, React will treat them as "new" on every render, which can cause unnecessary re-renders or effect executions.

Solution: We can use stable references (such as values that are guaranteed not to change between renders, e.g., primitives, or memoized functions) in the dependency array to avoid unnecessary re-executions.

Example: Using Stable References in the Dependency Array

```
import React, { useState, useEffect, useCallback } from 'react';

// Stable reference example
const App = () => {
  const [count, setCount] = useState(0);
  const [data, setData] = useState({ name: 'Neha Rathore' });

  // Memoizing the function to avoid re-creating on each render
  const updateData = useCallback(() => {
    setData(prev => ({
      ...prev,
      name: prev.name === 'Neha Rathore' ? 'Tushar Gupta' : 'Neha Rathore', // Using ternary to toggle name
    }));
  }, []); // No dependencies, this function will remain stable
  useEffect(() => {
   console.log('Effect triggered');
   // Only re-runs when `count` changes (stable references in the array)
  }, [count, updateData]); // updateData is stable because of useCallback

  return (
   <div>
    <p>Count: {count}</p>
    <button onClick={() => setCount(count + 1)}>Increment Count</button>
    <button onClick={updateData}>Update Data</button>
    <p>Data: {data.name}</p> { /* Printing the data */ }
   </div>
  );
};

export default App;
```

Behaviour:

- useCallback is used to memoize the updateData function, which ensures that the function reference remains stable across renders.
- In the useEffect, we only include count and updateData in the dependency array. React knows that updateData won't change unless explicitly defined, so it won't cause unnecessary re-executions of the effect.

Combining useCallback with useReducer for Action Dispatchers

In React, the combination of useCallback and useReducer can be a powerful pattern to optimize performance, especially when dealing with complex state management and actions in large components.

- **useReducer** is used for managing complex state logic in React components, often as a better alternative to useState when state updates are interdependent or require more intricate transitions.
- **useCallback** is used to memoize functions so that they are not recreated unnecessarily on every render, which can improve performance.

When to use this combination?

- When your component contains complex logic or multiple state transitions.
- When you need to optimize performance by passing functions down to child components or using functions in dependencies (useEffect, useMemo) without causing unnecessary re-renders or effect executions.

How does useReducer work?

useReducer is used when state transitions are complex or depend on multiple values. It works similar to Redux's reducer mechanism where an action is dispatched, and based on the action, the state is updated.

Example

```
const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

const App = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <div>
      <p>{state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
    </div>
  );
};
export default App;
```

Adding useCallback to optimize Action Dispatchers

When you pass dispatch functions down to child components or use them in useEffect, the function reference can change on every render, causing unnecessary re-renders. To avoid this, useCallback can be used to memoize the dispatch functions.

Example

```
import React, { useReducer, useCallback } from 'react';

// Initial state
const initialState = { count: 0 };

// Reducer function
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

const App = () => {
  // Using useReducer to manage the state
  const [state, dispatch] = useReducer(reducer, initialState);
  // Memoizing dispatch actions using useCallback
  const increment = useCallback(() => {
    dispatch({ type: 'increment' });
  }, [dispatch]); // Dependency array includes dispatch, ensuring it's not recreated

  const decrement = useCallback(() => {
    dispatch({ type: 'decrement' });
  }, [dispatch]);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};

export default App;
```

Why useCallback is useful here:

- **Prevents Function Re-Creation:** Without useCallback, the increment and decrement functions would be re-created on every render, even though their logic does not change.
- **Improved Performance:** When these functions are passed down to child components, React would treat them as new functions, causing unnecessary re-renders of those child components. Using useCallback avoids this issue by ensuring the function reference remains stable.
- **Stability in Dependency Arrays:** If the increment or decrement functions were used in useEffect or other hooks, their changing reference could cause the effect to run unnecessarily. Memoizing them ensures that the function reference remains stable, preventing unnecessary executions of effects.

Building a Custom Hook with Memoized Callbacks

When creating custom hooks in React, you may sometimes need to memoize callbacks that are used within the hook. Memoizing functions using useCallback ensures that the callback functions don't get recreated on every render, which can improve performance, especially when these functions are passed down as props to child components or are used in other hooks with dependencies.

Why Memoize Callbacks

1. **Performance Optimization**: In large applications, especially when dealing with child components or lists, frequent re-creations of callback functions can cause unnecessary re-renders.
2. **Stable References**: Memoizing ensures that a stable reference is maintained for callback functions, which is critical when the function is passed as a dependency to hooks like useEffect or useMemo.

Example

```
import { useState, useCallback } from 'react';
function useCounter(initialValue = 0) {    // custom hook
  const [count, setCount] = useState(initialValue);
    const increment = useCallback(() => { // Memoized increment callback
    setCount(prevCount => prevCount + 1);
  }, []);  // Empty dependency array ensures the function is memoized once.
  const decrement = useCallback(() => { // Memoized decrement callback
    setCount(prevCount => prevCount - 1);
  }, []);
  return { count, increment, decrement }; // Return  current count & memoized callbacks
}

function App() {
  const { count, increment, decrement } = useCounter(100);
  return (
    <div>      <h1>Count: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>     </div>
  );}
export default App;
```

Homework (to be submit on or before 5 December, 2024 (02:59:59 PM))

1. useState and useEffect

Exercise:

Build a simple **To-Do List app** with the following features:

- Add and delete tasks.
- Use useEffect to display a message whenever the task list changes (e.g., "Task added!" or "Task removed!").

Key Concepts:

- useState for managing the task list.
- useEffect for performing side effects based on state changes.

2. useContext

Exercise:

Create a **Theme Switcher** app with light and dark modes:

- Use useContext to manage and provide the theme state.
- Add a button to toggle between themes and apply the respective styles to components.

Key Concepts:

- Creating and using Context.
- Sharing state across components without prop drilling.

3. useReducer

Exercise:

Develop a **Shopping Cart** feature:

- Implement useReducer to manage cart actions (add item, remove item, update quantity, clear cart).
- Display the cart items with a total price calculation.

Key Concepts:

- Managing complex state using useReducer.
- Writing and handling reducer functions.

4. useCallback

Exercise:

Optimize a **Search Bar** with a list of items:

- Use useCallback to memoize a filter function that filters a list of items based on a search query.
- Demonstrate performance improvement by logging the re-render count of the child component displaying the filtered items.

Key Concepts:

- Preventing unnecessary renders.
- Memoizing functions using useCallback.

5. Combining Hooks

Exercise:

Create a **Blog Dashboard** with the following features:

- Display a list of blog posts fetched from an API.
- Provide a search bar (useCallback).
- Theme switching functionality (useContext).

Key Concepts:

- Combining multiple hooks to create a more realistic application.
- API calls using useEffect.