1. **Creating the example**: Now, let's create a simple example where we have a parent component with a counter and a child component that displays the counter. The parent component will pass a callback to the child component to increment the counter. We'll use `useCallback` to memoize the increment function.

   App.js:

   ```
   import React, { useState, useCallback } from 'react';
   import Counter from './Counter';

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

     // useCallback to memoize the increment function
     const increment = useCallback(() => {
       setCount(prevCount => prevCount + 1);
     }, []);

     return (
       <div className="App">
         <h1>Count: {count}</h1>
         <Counter increment={increment} />
       </div>
     );
   }

   export default App;
   ```

   Counter.js:

   ```
   import React from 'react';

   const Counter = React.memo(({ increment }) => {
     console.log('Counter component rendered');
     return (
       <button onClick={increment}>Increment</button>
     );
   });

   export default Counter;
   ```

   In this example:

   - The `App` component maintains a `count` state and has an `increment` function to increase the count.
   - The `increment` function is memoized using `useCallback` with an empty dependency array `[]`, meaning the function will only be created once and reused on subsequent renders.
   - The `Counter` component receives the `increment` function as a prop and renders a button that calls this function when clicked.
   - The `Counter` component is wrapped in `React.memo`, which is a higher-order component that memoizes the component itself, preventing it from re-rendering if its props haven't changed.

When you run this code, you'll notice that the `Counter` component does not re-render unnecessarily when the `App` component re-renders, because the `increment` function is memoized with `useCallback`.