memo, useMemo, useCallback

memo

As you know by now, if a parent component renders child component, and if any of the parent component's state or prop changes, it will trigger a re-rendering including all of the recursive child.

```
function Child() {
    console.log("Child component being rendered");

    return (
        <h1>This is the child component</h1>
    )
}

function Parent() {
    const [counter, setCounter] = useState(0);

    console.log("Parent component being rendered");

    return (
        <div>
            <p>This is the parent component {counter}</p>
            <button onClick={() => setCounter(counter + 1)}>Increments</button>
            <Child />
        </div>
    )
}



export default function App() {
    return (
        <Parent />
    )
}
```

That is given the above code snippet, if you click on the button to increment, it will trigger the following print out in the console:

```
Parent component being rendered
Child component being rendered
```

This should tell you that even though the `counter` state is not being used in the Child component, it still triggered a rendering because of the fact stated above.

Now what if you have a really expensive Child, and you don't want it to trigger a re-rendering if it isn't necessary? You can leverage `memo` which allow you to skip re-rendering of a component when it's props are unchanged. In this case we would like to memorize the Child and skip the re-rendering in the case that the parent component's state changes.

> Alternatively, and always preferred you should consider lifting your content up that is pointed out here React Parent and Child Component, which also accomplish the same thing, avoiding the re-rendering of the Child component in the case that the state of the Parent component changes.

How to use Memo

To use `memo` you just need to pass the Component that you wish to memorize into the call, it will return you the memorized version of your component and will usually not be re-rendered when its parent component is re-rendering.

> Do note that, React doesn't guarantee that the re-rendering will 100% not happen, it is a performance optimization not a guarantee.

In our example, if we want to skip the re-rendering we would just wrap our Child component in the call and then use it the same way as before.

```
function Child() {
    console.log("Child component being rendered");

    return (
        <h1>This is the child component</h1>
    )
}

const MemorizedChild = memo(Child);
```

```
function Parent() {
    const [counter, setCounter] = useState(0);

    console.log("Parent component being rendered");

    return (
        <div>
            <p>This is the parent component {counter}</p>
            <button onClick={() => setCounter(counter + 1)}>Increments</button>
            <MemorizedChild />
        </div>
    )
}


export default function App() {
    return (
        <Parent />
    )
}
```

Now if you click on increment, only the Parent component will re-render the Child component will not because there are no prop changes that will result in the Child needing to be updated.

Second parameter

`memo(Component, arePropsEqual)`

If you take a look at `memo` you will see that there is a second parameter that you can provide for the memorization call.

The second parameter is a function that accepts two arguments, the component's previous props, and its new props. It should return `true` if the old and new props are equal: Which means the component will not be re-rendered, otherwise, it should return `false`, indicating that the function should be re-rendered.

This second argument gives you more control on exactly when to expire the old component and re-render the memorized component.

useMemo / useCallback

Going to be grouping these two together because they are really similar, `useCallback` is actually a syntactic sugar for `useMemo`.

By definition:

- `useMemo`: Caches the result of a calculation between re-renders
- `useCallback`: Caches a function definition between re-renders

They have similar function signature being:

```
const cached = useMemo(calculateValue, dependencies)
const cached = useCallback(fn, dependencies)
```

- calculateValue: **Being the function that computes the final value that you're aching. It takes in no argument and should return a value of any type.** On the first render, React will call this function to retrieve the initial value, on next render, React will check if any of the dependencies have changed since the last render, if it hasn't it will return the same value, otherwise, it will call the function again and give you the new value and caching it
- fn: Similar idea, instead of value, it will be the function that you would want to cache. **This function can take in any number of arguments and return any value.** React during the first render will return this function back to you, then on the next render, React will check if any of the dependencies have changed since the last render, if it hasn't it will return you the same function, otherwise, it will return you the updated function.

> With newer React compiler, the optimization is actually done for you automatically hence you don't need to call these yourself anymore

Using useMemo and useCallback

With `useMemo` a great use case of this is to cache an expensive operation. Say you need to sort through a list and that list is huge but it doesn't change often, you would 100% want to leverage `useMemo` to cache the result of the sort to avoid re-calculating the same value when the component is re-rendered, otherwise your application might not be responsive to the user.

```
function sort(array) {
  // expensive sorting
}

const cached = useMemo(() => sort(input), [input]);
```

With `useCallback`, like it was mentioned, it is just a syntactic sugar on `useMemo`, `useMemo(() => helloWorld, [])` is functionallity

```
function helloWorld() {
}

useMemo(() => helloWorld, []); // Returns you a function as value that you would want to cache
useCallback(helloWorld, []); // Takes in the function as value that you want to cache, they
are equivalent
```

`useCallback` is especially useful for skipping re-rendering of components.

Say we again have the same Parent and Child component relationship again.

```
function ProductPage({ productId, referrer, theme }) {
  // Every time the theme changes, this will be a different function...
  function handleSubmit(orderDetails) {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }

  return (
    <div className={theme}>
      {/* ... so ShippingForm's props will never be the same, and it will re-render every time
*/}
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

Don't worry about the actual working of the Component, focus on the `handleSubmit` function. Now in Javascript, whenever you run `function() {}` or `() => {}` this always triggers a new reference being created, that means, whenever `ProductPage` is being rendered (when any of its prop or state changes), `handleSubmit` will be re-created with a new definition.

And that means `ShippingForm` will be re-rendered as well because the prop changed. Now this is where `useCallback` comes in handy, because it caches the function result for you allowing `ShippingForm` to skip re-rendering if you combine it with the memo technique that was discussed.

If you **only** use memo it will not work because like was mentioned, `function() {}` or `() => {}` triggers a new function definition every time it is encountered. So if you want to skip re-rendering of the Child then you will want to do both, `useCallback` to cache the function, then use `memo` to skip re-rendering of the Child component because the prop didn't change.

```
import { memo, useCallback, useState } from "react";



function Child({onSubmit}) {
    console.log("Child component being rendered");

    return (
        <div>
            <h1>This is the child component</h1>
            {onSubmit()}
        </div>
    )
}

const MemorizedChild = memo(Child);



function Parent({theme}) {
    const [counter, setCounter] = useState(0);

    console.log("Parent component being rendered");

    const cached = useCallback(() => {
        return (
            <h1>Hello world {counter}</h1>
        )
    }, [counter])

    return (
        <div>
            <p>This is the parent component {counter}</p>
            <button onClick={() => setCounter(counter + 1)}>Increments</button>
            <MemorizedChild onSubmit={cached} />
        </div>
    )
```

```
}


export default function App() {
    const [theme, setTheme] = useState("dark");

    return (
        <div>
            <Parent theme={theme}/>
            <button onClick={() => setTheme(theme === "dark" ? "light" : "dark")}>Toggle
Theme</button>
        </div>
    )
}
```

Revision #1
Created 2026-01-31 21:28:04 UTC by Tamarine
Updated 2026-01-31 22:09:48 UTC by Tamarine