**🧱 Step 1: What is React.memo (super simple)**

Imagine you have a **Parent component** and a **Child component**.

Normally —  
👉 When **Parent** re-renders, all its **Children** re-render too  
(even if their props didn’t change!)

That’s wasteful, right?

React.memo() tells React:

“Hey React! If this child’s props haven’t changed, don’t re-render it again.”

So it **remembers (memoizes)** the last rendered result of the child.

**🧩 Example:**

function Child({ name }) {

console.log("Child rendered");

return <p>Hello {name}</p>;

}

const MemoChild = React.memo(Child);

function App() {

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

return (

<div>

<button onClick={() => setCount(count + 1)}>Increase Count</button>

<MemoChild name="Veda" />

<p>Count: {count}</p>

</div>

);

}

🧠 Output in console:

Child rendered

(only once, at the beginning)

When you click the button —  
App re-renders, but MemoChild **doesn’t re-render** because its prop (name="Veda") didn’t change. ✅

That’s **React.memo**.

**🧠 Step 2: Problem with functions**

Now imagine we pass a **function** as a prop:

<MemoChild onClick={() => console.log("Clicked")} />

Even though the function looks same,  
👉 it’s **new every render**, so React thinks prop changed,  
and **re-renders child** every time 😩

That’s where **useCallback** helps!

**⚙️ Step 3: What is useCallback**

useCallback says:

“Please remember this function and don’t recreate it unless something important changes.”

So we write:

const handleClick = useCallback(() => {

console.log("Clicked");

}, []);

Now the function stays **same between re-renders** ✅  
and if we pass it to a memoized child, that child **won’t re-render unnecessarily.**

**🧮 Step 4: What is useMemo**

useMemo is like useCallback, but for **values**, not functions.

Example:

const filteredList = useMemo(() => {

return products.filter(p => p.name.includes(search));

}, [products, search]);

If search or products don’t change, React **reuses** the old filteredList value —  
so no extra filtering work each time ✅

**🎯 Step 5: Side-by-side Example (super easy)**

Let’s put all 3 together now — React.memo, useMemo, and useCallback.

import React, { useState, useMemo, useCallback } from "react";

const ProductList = React.memo(({ products, onSelect }) => {

console.log("🧩 ProductList rendered");

return (

<ul>

{products.map((p) => (

<li key={p.id} onClick={() => onSelect(p.name)}>

{p.name}

</li>

))}

</ul>

);

});

export default function App() {

const [search, setSearch] = useState("");

const [theme, setTheme] = useState("light");

const products = [

{ id: 1, name: "Veda Phone" },

{ id: 2, name: "Reva Laptop" },

{ id: 3, name: "Meda Watch" },

];

// 🧠 useMemo → remembers filtered products

const filteredProducts = useMemo(() => {

console.log("🔍 Filtering...");

return products.filter((p) =>

p.name.toLowerCase().includes(search.toLowerCase())

);

}, [search]);

// ⚙️ useCallback → remembers function reference

const handleSelect = useCallback((name) => {

alert(`You selected ${name}`);

}, []);

return (

<div>

<h3>Theme: {theme}</h3>

<button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>

Change Theme

</button>

<br />

<input

placeholder="Search product..."

value={search}

onChange={(e) => setSearch(e.target.value)}

/>

<ProductList products={filteredProducts} onSelect={handleSelect} />

</div>

);

}

**🔍 Let’s understand the flow**

1. **First render**
   * Filters run → 🔍 Filtering...
   * ProductList renders → 🧩 ProductList rendered
2. **You type in search box**
   * search changes → useMemo runs again → filtering done again  
     ✅ That’s expected.
3. **You change theme**
   * theme changes → App re-renders  
     ❌ But search and handleSelect didn’t change →  
     so:
   * useMemo skips re-filtering
   * useCallback keeps same function
   * ProductList doesn’t re-render ✅ (super optimized!)

**🧠 In short:**

| **Hook** | **What it remembers** | **Why used** | **Example** |
| --- | --- | --- | --- |
| React.memo | Whole component | Stop re-render if props same | <MemoChild /> |
| useCallback | A **function** | Prevent new function creation | handleClick |
| useMemo | A **calculated value** | Prevent re-calculation | filteredProducts |

Would you like me to show the **same example without optimization** (to see how many times it re-renders and filters again)?  
That way you’ll instantly *see the difference* in console logs.

**You said:**

it means we have to use these 3 are always together

**ChatGPT said:**

💯 Excellent question — and this is the **most common confusion** developers have!  
Let’s make it **crystal clear** 👇

**🚫 NO — we don’t always need to use all 3 (React.memo, useMemo, useCallback) together.**

They’re **related**, but used for **different optimization purposes**.  
You use them **only when** there’s a real need — not by default.

**⚙️ Think of them like a family:**

| **Hook** | **Type** | **Role** |
| --- | --- | --- |
| 🧩 React.memo | Component wrapper | Stops **child** from re-rendering if its props haven’t changed |
| 🧠 useMemo | Value memoizer | Remembers **expensive calculations / filtered data** |
| ⚙️ useCallback | Function memoizer | Remembers **function references** passed as props |

**🎯 When to use each (with real-life examples):**

**🧩 React.memo → *Use when:***

You have a **child component** that receives props and is re-rendering even when props don’t change.

📱 Example:

<Child name="Veda" />

If the parent re-renders 10 times but name never changes,  
you can wrap the child in React.memo() to stop wasteful re-renders.

✅ *Only needed for reusable child components.*

**⚙️ useCallback → *Use when:***

You pass **functions as props** to memoized child components.

📞 Example:

const handleClick = useCallback(() => {

console.log("Clicked");

}, []);

If you don’t memoize this function, React thinks it’s a new one every render →  
child re-renders unnecessarily.

✅ *Only needed when passing functions to memoized children.*

**🧠 useMemo → *Use when:***

You’re doing **expensive calculations** (like filtering, sorting, or computing totals).

🧮 Example:

const filteredData = useMemo(() => {

return products.filter(p => p.name.includes(searchTerm));

}, [products, searchTerm]);

✅ *Only needed for performance-heavy calculations.*

**💡 Summary — When you’d use them together**

| **Situation** | **What you’d use** |
| --- | --- |
| Big dataset filtering | useMemo |
| Child component keeps re-rendering unnecessarily | React.memo |
| You’re passing a callback (function) prop to that memoized child | useCallback |

So yes — sometimes they **all appear together**,  
but only because that’s a case where all 3 are **relevant** to optimize different parts of performance.

**🚀 Example analogy:**

Think of them like a car’s features:

* React.memo → brakes (stops useless re-renders)
* useCallback → cruise control for functions (keeps them stable)
* useMemo → fuel saver (avoids extra computation)

You don’t always drive with all 3 ON.  
You use what’s needed based on the road you’re on. 🛣️

**🧩 Why React sees your function as a “new value” on every render**

Every time your component **re-renders**, React **runs the function again** from top to bottom — like pressing the “restart” button.

So in this code:

function ProductPage() {

const handleClick = () => {

console.log("Clicked!");

};

return <button onClick={handleClick}>Click</button>;

}

When React re-renders ProductPage,  
it creates a **brand new handleClick function** again in memory. ⚙️

Even though the code looks the same — under the hood, it’s a **different function object** in memory each time.

**🧠 Let’s visualize**

Imagine this happens:

| **Render** | **handleClick** | **Memory ID** |
| --- | --- | --- |
| 1st render | () => console.log("Clicked!") | 🧠 #101 |
| 2nd render | () => console.log("Clicked!") | 🧠 #202 |
| 3rd render | () => console.log("Clicked!") | 🧠 #303 |

They all look identical…  
but React sees: “Oh! The function reference changed (101 → 202 → 303). Must update the prop.”

So if you pass it to a **child component**, React thinks that prop has changed →  
Child re-renders unnecessarily 😩

**🧠 Solution → useCallback**

Now if you wrap it like this 👇

const handleClick = useCallback(() => {

console.log("Clicked!");

}, []);

React will **remember** that function between renders.

So now it behaves like:

| **Render** | **handleClick** | **Memory ID** |
| --- | --- | --- |
| 1st render | same function | 🧠 #101 |
| 2nd render | reuses same one | 🧠 #101 |
| 3rd render | still the same | 🧠 #101 |

✅ React now says — “Prop didn’t change!” → no useless re-render in child.

**🧮 So to summarize:**

| **Concept** | **Without useCallback** | **With useCallback** |
| --- | --- | --- |
| Function recreated on every render | ✅ Yes | ❌ No |
| Function memory reference stable | ❌ No | ✅ Yes |
| Child component re-renders due to prop change | ✅ Yes | ❌ No |
| Performance | Slower on large trees | Optimized |

**🧩 When do you really need it?**

✅ Use it when:

* You pass the function **as a prop** to a memoized child (React.memo).
* The child **doesn’t need to re-render** every time.

🚫 Don’t use it for every single function — only for ones that cause re-render issues.  
(Otherwise, it just adds unnecessary complexity.)

**💡 Why we use useCallback:**

👉 Every time a React component **re-renders**, all its functions are **recreated in memory** — even if their logic didn’t change.  
So React sees them as **new functions** every render.

This can cause:

* Extra **memory usage**
* **Unnecessary re-renders** in child components (especially if the function is passed as a prop)

**🧠 What useCallback does:**

useCallback **stores** (memoizes) the function —  
and gives you back **the same function reference** between renders,  
**until** the dependencies change.

**📦 Example:**

const handleClick = () => {

console.log("clicked");

};

If the component re-renders 10 times — React will create 10 new handleClick functions in memory 😅  
Even though they all *do the same thing*.

But with:

const handleClick = useCallback(() => {

console.log("clicked");

}, []);

➡️ Now React will **reuse** the same function across renders.  
So it **saves memory** and **prevents child components** from re-rendering unnecessarily.

**🧩 In short:**

| **Without useCallback** | **With useCallback** |
| --- | --- |
| New function created on every render | Same function reused (memoized) |
| More memory usage | Less memory usage |
| Causes unnecessary re-renders in children | Prevents unnecessary re-renders |
| Good for small components only | Best for optimized / nested UIs |
|  |  |