Controlled vs Uncontrolled Hooks in React

Controlled Hooks

Controlled components/hooks rely on **state** maintained by the **parent or React itself**. The state is explicitly **controlled via `useState`, `useReducer`, or props**.

Hook	Type	Description
`useState`	Controlled	Manages internal component state (form inputs, UI toggles, etc).
`useReducer`	Controlled	Similar to `useState` but for complex state logic (e.g., global state, reducers).
`useEffect`	Controlled	Executes side effects based on dependency array.
`useMemo`	Controlled	Memoizes values based on dependencies you control.
`useCallback`	Controlled	Memoizes functions — you control when it's re-created.
`useRef` *(as state holder)*	Controlled	If used to persist controlled data across renders (e.g. counterRef.current = value).
`useContext`	Controlled	Consumes values from a `Context.Provider` — which is explicitly provided.
`useForm` (from libs like React Hook Form)	Controlled	Form state is explicitly handled and synchronized with component state.

Example – Controlled Input

```
const [name, setName] = useState("");

<input value={name} onChange={(e) => setName(e.target.value)} />
```

- You control the `value` with `state`.

Uncontrolled Hooks / Behaviors

Uncontrolled means the data is **managed by the DOM or refs**, not by React's state. This can be faster and simpler when you don't need fine-grained control.

Hook	Type	Description
`useRef` *(as DOM reference)*	Uncontrolled	Ref to access DOM nodes or mutable values without causing re-renders.

Hook	Type	Description
`useImperativeHandle`	Uncontrolled	Exposes imperative methods in custom components using `ref`.
`forwardRef`	Uncontrolled	For exposing refs to children (e.g., custom input wrappers).
Native form inputs without state	Uncontrolled	Uses `defaultValue` or DOM access, not `state`.
React Hook Form *(with uncontrolled mode)*	Uncontrolled	Registers fields with `ref` and tracks data internally.

📌 Example – Uncontrolled Input

```
const nameInputRef = useRef();

<input ref={nameInputRef} />

<button onClick={() => alert(nameInputRef.current.value)}>
  Submit
</button>
```

- You never use `useState` here. The DOM manages the input value.

🎯 When to Use What?

Scenario	Recommended Hook Type
You need real-time validation	☑ Controlled (`useState`)
You don't care about input until submit	☑ Uncontrolled (`useRef`)
Performance-critical large forms	☑ Uncontrolled (React Hook Form w/ `ref`)
Dynamic UI behavior	☑ Controlled
Simple toggle logic	☑ Controlled (`useState`)
DOM measurements	☑ Uncontrolled (`useRef`)
Scroll control / focusing input	☑ Uncontrolled (`useRef`)

⚖️ Summary Table

Hook	Controlled ☑	Uncontrolled ☑	Notes
`useState`	☑	❌	React state-driven
`useReducer`	☑	❌	Complex state logic

Hook	Controlled ☑	Uncontrolled ☑	Notes
`useEffect`	☑	❌	Controlled by deps
`useRef` (state use)	☑	☑	Dual role: data or DOM
`useImperativeHandle`	❌	☑	Ref-based logic
`forwardRef`	❌	☑	Ref passing
`useContext`	☑	❌	Controlled by provider
`useCallback` / `useMemo`	☑	❌	Controlled by deps
React Hook Form	☑ / ☑	☑	Supports both modes

🧒➡️👪 Can a Child Change Parent's State?

- ❌ **No**, a child **cannot directly** change a parent's state.
- ☑ **Yes**, a child can **indirectly** cause the parent to change its state by calling a **function passed via props** from the parent.

❌ Wrong Approach (Direct Access Attempt)

```jsx
// ParentComponent.jsx
import React, { useState } from 'react';
import Child from './Child';

function Parent() {
  const [count, setCount] = useState(0);
  return <Child parentState={count} />;
}

export default Parent;

// Child.jsx
const Child = ({ parentState }) => {
  // ❌ This does NOT work and will throw error
  parentState = 10; // ❌ Trying to mutate a prop (React will warn)
  return <div>{parentState}</div>;
};
```

❌ What's wrong here?

- Props are **read-only** in React.
- Child trying to **mutate** a value (`parentState`) it received from parent — illegal ❌.
- React will show a warning: `Cannot assign to read only property`.

☑ Correct Approach (Indirect State Change)

```jsx
// ParentComponent.jsx
import React, { useState } from 'react';
import Child from './Child';

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

  // ☑ Pass the setter function to child
  const incrementCount = () => setCount(count + 1);

  return (
    <>
      <h2>Count: {count}</h2>
      <Child onIncrement={incrementCount} />
    </>
  );
}

export default Parent;

// Child.jsx
const Child = ({ onIncrement }) => {
  return <button onClick={onIncrement}>➕ Increment from Child</button>;
};
```

☑ What's right here?

- Parent owns the state.
- Parent passes a function (`onIncrement`) to child via props.
- Child **calls** the function when needed.
- Child doesn't know **how** the state is managed — just that it can trigger it. 🎯

🎓 Best Practices

☑ Do	❌ Don't
Pass callback functions from parent to child	Try to mutate parent state directly in child
Keep state lifting centralized (usually parent)	Create duplicate state in child and parent for the same thing
Use prop drilling or context for deeply nested updates	Use side effects to indirectly mutate parent

---

🎁 Bonus Minimal Reusable Pattern

```jsx
const Parent = () => {
  const [msg, setMsg] = useState("Hello");

  const updateMsg = (newMsg) => setMsg(newMsg);

  return (
    <>
      <h2>{msg}</h2>
      <Child onUpdate={updateMsg} />
    </>
  );
};

const Child = ({ onUpdate }) => (
  <button onClick={() => onUpdate("👋 Updated from Child")}>
    Update Message
  </button>
);
```

🧠 Summary

- Props = One-way data flow ⬇️
- State = Local to component 💾
- To "lift" state control: ☑️ pass a handler
- Never: ❌ mutate props