2620. Counter

Easy

Leetcode Link

Problem Description

The problem requires us to write a function that generates a "counter" function. When the counter function is first called, it should return an integer `n` which is given as input when creating the counter. On each subsequent call, the counter function should return an integer that is one greater than the value it returned on the previous call. Essentially, it keeps track of a count, starting from `n` and increasing by 1 each time it is invoked.

Intuition

Our goal is to retain the state between function calls. To achieve this, we can use closure properties in the JavaScript-related TypeScript language. A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain. The closure has three scope chains: it has access to its own scope, the outer function's variables, and the global variables.

The enclosed variable `i` is initialized with the value of `n` and is kept alive after the `createCounter` function execution context is popped off the call stack. In other words, every time the inner function is called, it has access to the `i` variable that was defined in the same context.

The intuition behind the solution is:

- Define a variable `i` inside the `createCounter` function's scope and set its initial value to `n`.
- Return an inner function that increments and returns the variable `i` when called.
- Since each call to the inner function will retain the reference to `i`, it will keep incrementing `i` by 1 each time without resetting it.

The elegant part of the solution is utilizing the closure's ability to maintain state across multiple calls without the need for an external state manager or object.

Solution Approach

The implementation of the solution is straightforward due to TypeScript's support for closures—a key concept in functional programming.

Here is the step-by-step approach of the solution:

- First, we define a function `createCounter` that takes an integer `n` as an argument.
- Within `createCounter`, we initialize a variable `i` with the value of the argument `n`. This variable `i` will serve as our counter state.
- We then define and return an inner function—this is where our closure comes into play. This inner function has access to `i` because `i` is in the scope of the outer function `createCounter`.
- In this inner function, we increase the value of `i` by 1 using the increment operator `++`. This operator increases `i` by one and then returns the original value of `i` before the increment.
- As a result, every time the returned inner function is called, it will increment `i` and then return its current value.

The key algorithmic concept at work here is the use of a closure that captures the local variable `i` from its surrounding scope in `createCounter`, allowing the inner function to have a persistent state across its invocations. This pattern effectively creates a private counter variable that is only accessible and modifiable by the returned function.

The important thing to note is that TypeScript, like JavaScript, treats functions as first-class citizens, meaning that functions can be assigned to variables, passed as arguments to other functions, and can be returned from functions as well. This feature is what enables us to create and return the inner function that acts as our counter.

The corresponding TypeScript code implementing the solution would be:

```typescript
function createCounter(n: number): () => number {
    let i = n;
    return function () {
        return i++;
    };
}
```

By calling `createCounter(10)`, we create a counter that starts at 10. Each call to the returned function will then provide the next number in the sequence (10, 11, 12, and so on), showcasing the closure's ability to maintain and update the counter state.

Example Walkthrough

Let's illustrate the solution approach with a simple example. We'll use `createCounter` to generate a counter that starts at 5 and then call the counter function a few times to see how it increments the internal count.

1. We call `createCounter(5)`, which initializes the variable `i` to 5 within its scope and sets up the closure. This call returns a new function—let's call it `myCounter`.
2. We call `myCounter()` for the first time. Since `i` was initialized to 5, the function increments `i` to 6 but returns the original value of `i`, which is 5.
3. We call `myCounter()` again. This time, `i` starts at 6, increments to 7, and then returns 6.
4. We call `myCounter()` a third time. Now, `i` starts at 7, increments to 8, and then returns 7.

Here's how that would look in code:

```typescript
// Step 1: We create the counter starting at 5.
let myCounter = createCounter(5);

// Step 2: We call the counter for the first time.
console.log(myCounter()); // Output: 5

// Step 3: We call the counter for the second time.
console.log(myCounter()); // Output: 6

// Step 4: We call the counter for the third time.
console.log(myCounter()); // Output: 7

// If we keep calling myCounter(), it will continue to output the next integer in the sequence.
```

Each call to `myCounter()` persistently increases the variable `i` by 1. Despite multiple invocations, the counter retains its current value thanks to the closure capturing the `i` variable. This behavior allows `myCounter` to act like a true "counter", demonstrating the effective use of closures in TypeScript to maintain state between function invocations.

Python Solution

```python
def create_counter(initial_count):
    """
    Creates a counter function that, when called, will return a number that increments by 1
    each time the counter function is called.

    :param initial_count: The starting point for the counter
    :return: A function that when invoked, returns an incremented number
    """
    # Variable 'current_count' holds the current state of the counter
    current_count = initial_count

    # The returned function encapsulates the 'current_count' variable.
    # Each time it is called, it increments 'current_count' by one, then returns the new value.
    def counter():
        nonlocal current_count
        current_count += 1
        return current_count - 1

    return counter

# Example usage:
# Create a counter starting at 10
counter = create_counter(10)
print(counter())  # Should output: 10
print(counter())  # Should output: 11
print(counter())  # Should output: 12
```

Java Solution

```java
import java.util.function.IntSupplier;

/**
 * Creates a counter function that, when called, will return a number that
 * increments by 1 each time the counter function is called.
 *
 * @param initialCount The starting point for the counter.
 * @return An IntSupplier that when invoked, returns an incremented number.
 */
public class CounterCreator {

    public static IntSupplier createCounter(int initialCount) {
        // An array is used to hold the current state of the counter, as we need a final or effectively final variable
        // to be used in the lambda's closure, but we need to alter it.
        final int[] currentCount = {initialCount};

        // The lambda expression encapsulates the 'currentCount' variable. Each time it is called,
        // it increments the value at currentCount[0] by one, then returns the updated value.
        return () -> currentCount[0]++;
    }

    // Example usage:
    public static void main(String[] args) {
        // Create a counter starting at 10
        IntSupplier counter = createCounter(10);
        System.out.println(counter.getAsInt()); // Should output: 10
        System.out.println(counter.getAsInt()); // Should output: 11
        System.out.println(counter.getAsInt()); // Should output: 12
    }
}
```

C++ Solution

```cpp
#include <iostream>
#include <functional>

/**
 * Creates a counter function that, when called, will return a number that
 * increments by 1 each time the counter function is called.
 *
 * @param initialCount The starting point for the counter.
 * @return A function object (lambda) that, when invoked, returns an incremented number.
 */
std::function<int()> createCounter(int initialCount) {
    // Variable 'currentCount' holds the current state of the counter.
    int currentCount = initialCount;

    // The returned function object (lambda) encapsulates the 'currentCount' variable.
    // Each time it is called, it increments the value of 'currentCount' by one,
    // then returns the updated value.
    // Captures 'currentCount' by reference so that the counter state persists across calls.
    return [&currentCount]() -> int {
        return currentCount++;
    };
}

// Example usage:
int main() {
    // Create a counter starting at 10.
    auto counter = createCounter(10);

    // The counter retains its state across calls because 'currentCount' is captured by reference.
    std::cout << counter() << std::endl; // Should output: 10
    std::cout << counter() << std::endl; // Should output: 11
    std::cout << counter() << std::endl; // Should output: 12

    return 0;
}
```

Typescript Solution

```typescript
/**
 * Creates a counter function that, when called, will return a number that
 * increments by 1 each time the counter function is called.
 *
 * @param {number} initialCount - The starting point for the counter.
 * @returns {function} A function that when invoked, returns an incremented number.
 */
function createCounter(initialCount: number): () => number {
    // Variable 'currentCount' holds the current state of the counter.
    let currentCount = initialCount;

    // The returned function encapsulates the 'currentCount' variable. Each time it is called,
    // it increments the value of 'currentCount' by one, then returns the updated value.
    return function (): number {
        return currentCount++;
    };
}

// Example usage:
// Create a counter starting at 10.
const counter = createCounter(10);
console.log(counter()); // Should output: 10
console.log(counter()); // Should output: 11
console.log(counter()); // Should output: 12
```

Time and Space Complexity

Time Complexity: The time complexity of the counter function is `O(1)`, which is a constant time operation. Every time the function is called, it performs one increment operation and returns the new value.

Space Complexity: The space complexity of `createCounter` can also be considered `O(1)` since it creates a single closure with a variable `i` that maintains the state, no matter how many times the counter function is called. The counter itself does not allocate any additional space that is dependent on the input size `n` or the number of times it's called.