

Inductive Definition to Recursive Function

Factorial Function

Consider the following recursive definition of the factorial function.

$$n! = \begin{cases} 1, & \text{if } n = 0, \\ n \times (n - 1)!, & \text{if } n > 0. \end{cases}$$

The function is used to define itself. The definition is an **equation** with a **computational counterpart**.

The Equation

The factorial function satisfies the functional equation^a.

$$F(n) = \begin{cases} 1, & \text{if } n = 0, \\ n \times F(n - 1), & \text{if } n > 0. \end{cases}$$

^aThe factorial is the fixed-point of this equation

Computation of 4!

$$\begin{aligned}4! &= 4 \times 3! \\ &= 4 \times (3 \times 2!) \\ &= 4 \times (3 \times (2 \times 1!)) \\ &= 4 \times (3 \times (2 \times (1 \times 0!))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))) \\ &= 4 \times (3 \times (2 \times 1)) \\ &= 4 \times (3 \times 2) \\ &= 4 \times 6 = 24\end{aligned}$$

Note

- There is no value computation in the first four steps. The function is being **unfolded**.
- The value computation starts only after the **basis** of the definition is reached.
- Last four steps computes the values.

A function in C Language may call itself

A function that calls itself directly or indirectly is called a **recursive function**. Unfolding and delayed computation can be simulated by such a function.

Recursive Call

If a function calls itself, the obvious question is about the termination of the process.

$$A() \xrightarrow{\text{call}} A \quad A() \xrightarrow{\text{call}} A \quad A() \xrightarrow{\text{call}} A \quad A() \dots$$

Recursive Call

Naturally the call cannot be **unconditional**. The **basis** of an inductive (recursive) definition provides the condition for termination. The function calls itself to reach the termination condition, the basis.

Useless for Computation

The factorial function also satisfies the following equation,

$$F(n) = \begin{cases} 1, & \text{if } n = 0, \\ \frac{F(n+1)}{n+1}, & \text{if } n > 0. \end{cases}$$

but cannot be used for computation as a call to itself does not reach the basis.

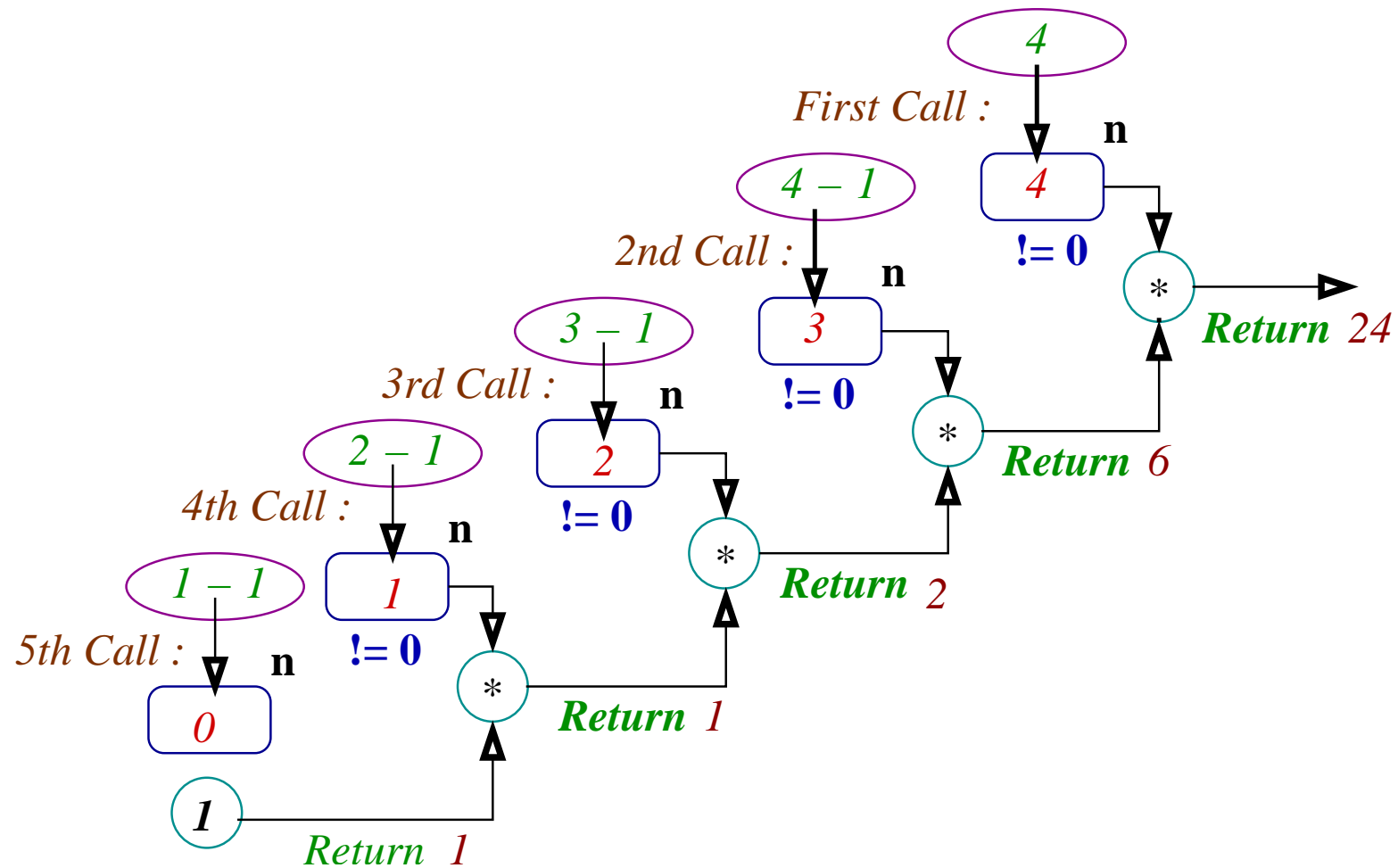
Recursive factorial Function

```
int factorial(int n)
{
    if (n == 0) return 1 ;
    else return n*factorial(n - 1) ;
} // factorialFR1.c
```

The function takes an actual parameter p (a non-negative integer) and returns the value of $p!$.

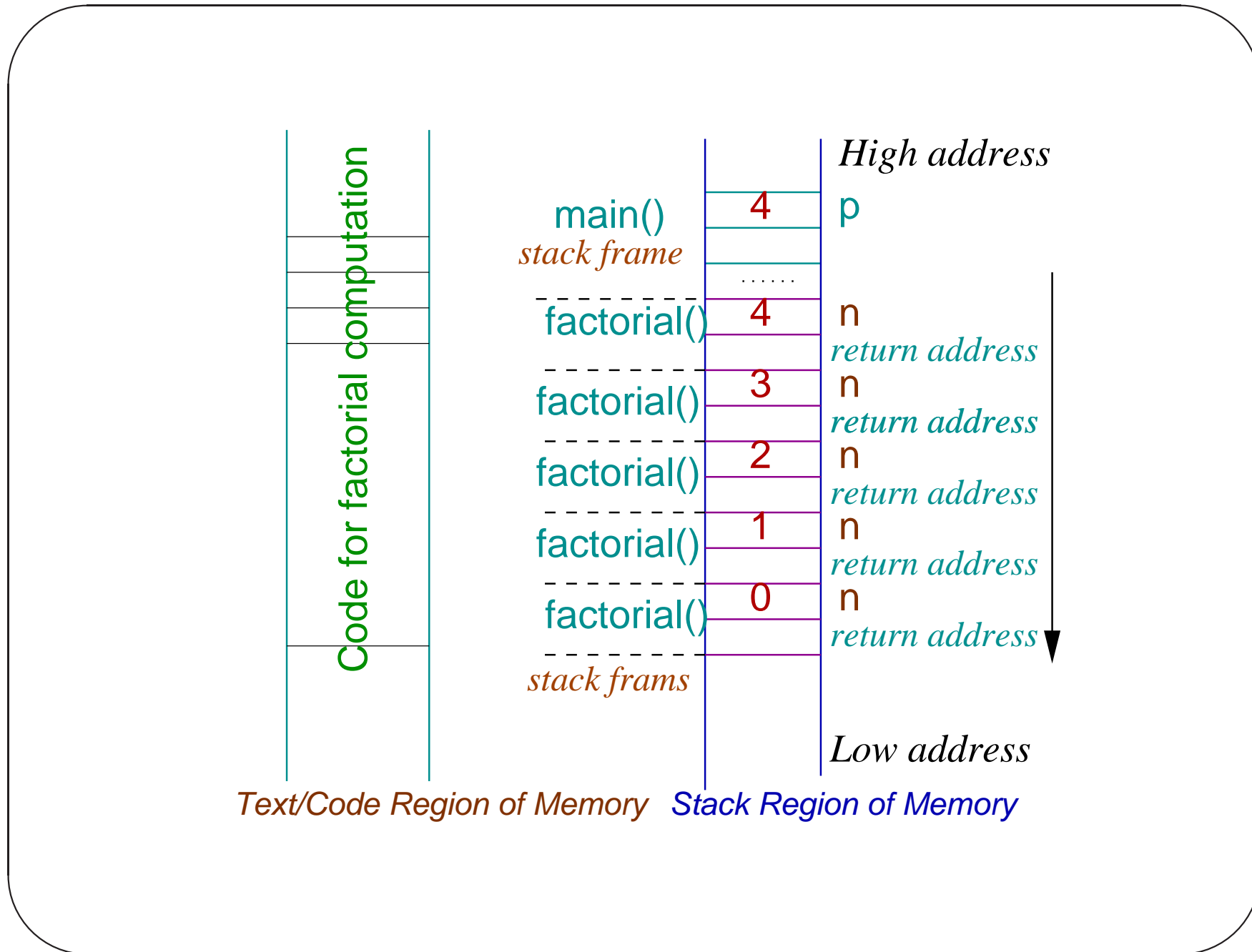
Different Calls and Incarnations of n

Actual Parameter is 4



Same Code Different Data

- Same code is used for every recursive call.
- Data (local) is different in every recursive call.



Note

- The first phase does not compute the value but **unfolds** the recursion upto the base case.
- The value computation starts from the base case.

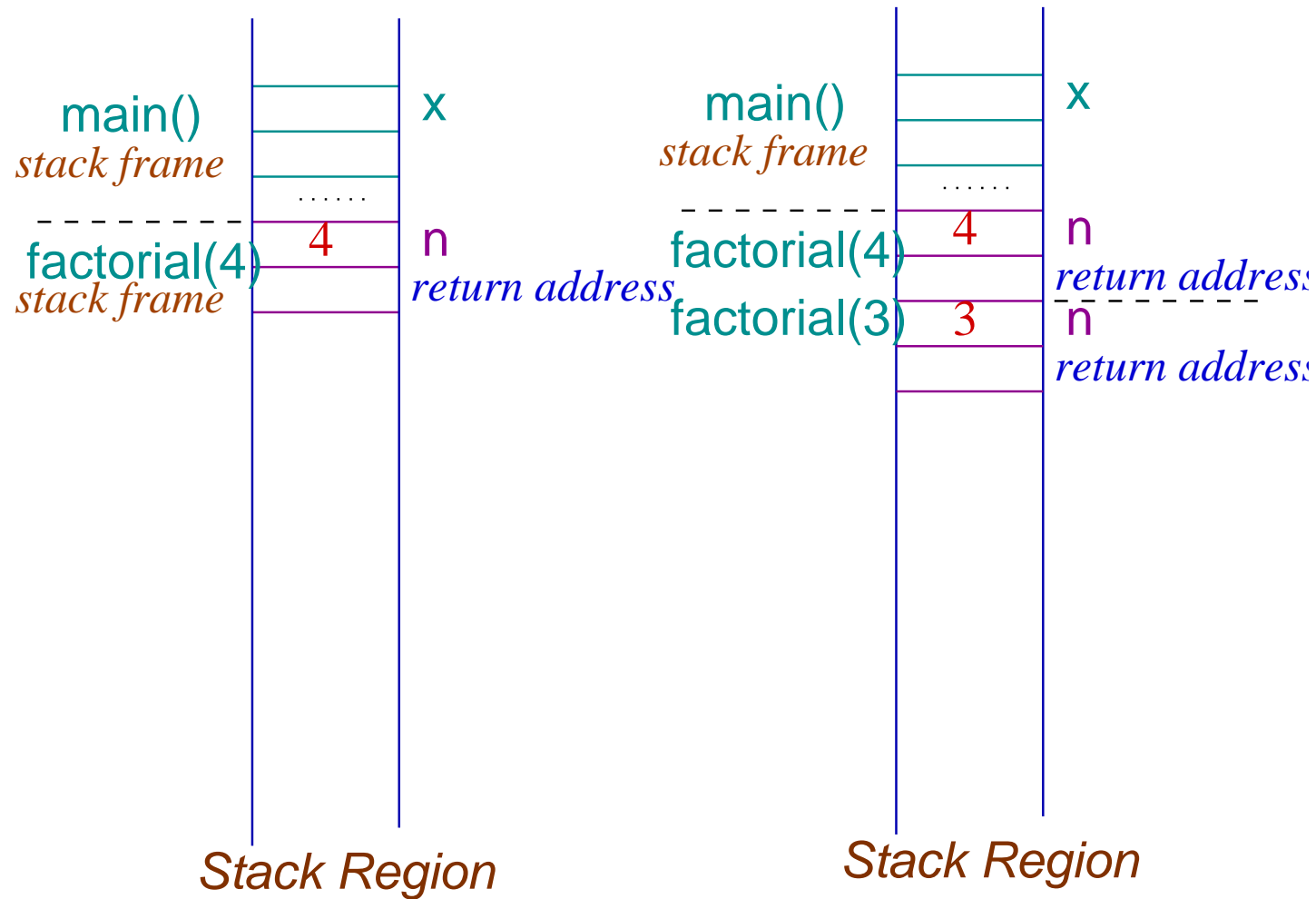
Note

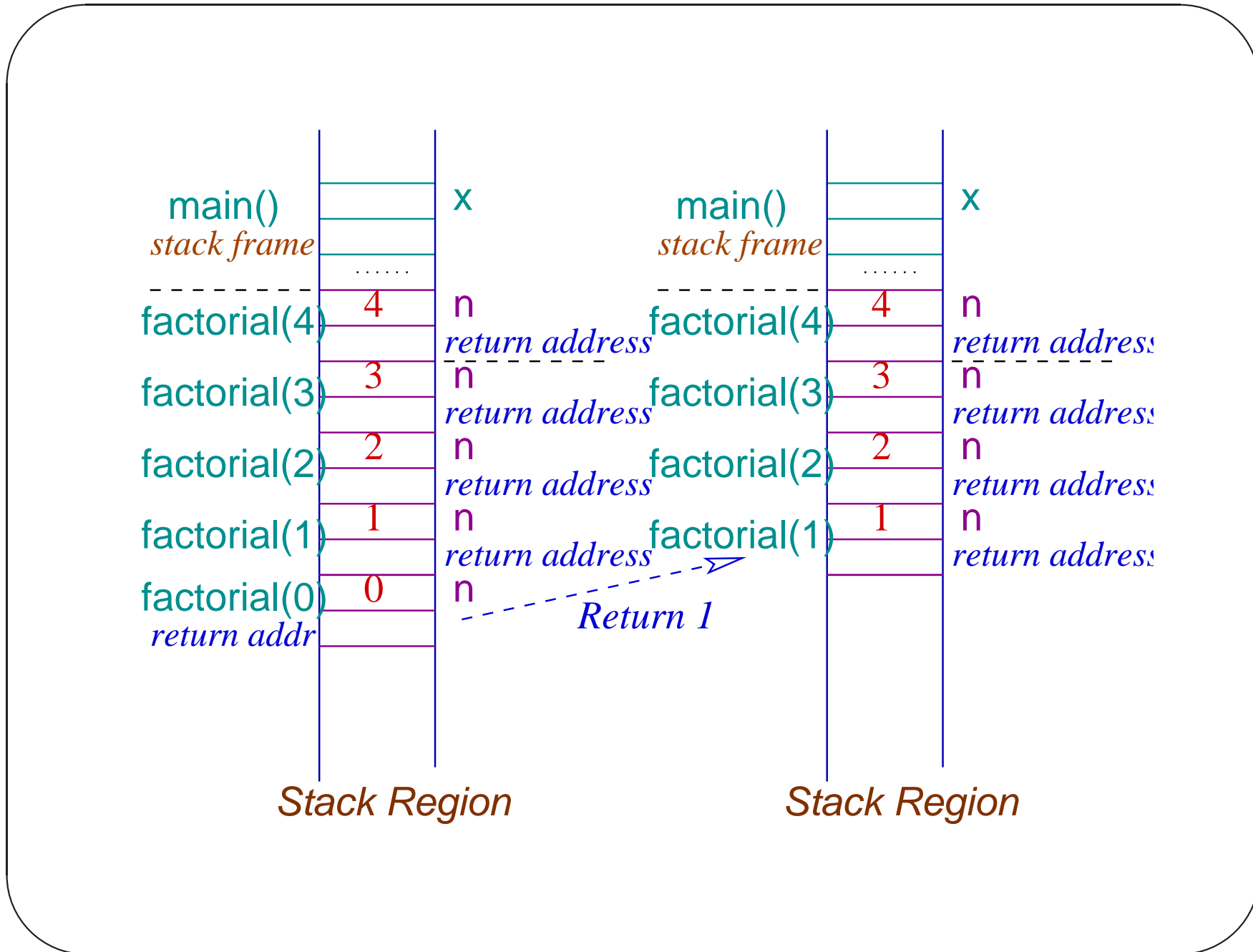
- For every call there is new **incarnation** of all the **formal parameters** and the **local variables** (that are not **static**). The variable names get bind to different memory locations.
- Variables of one invocation are not visible from another invocation.

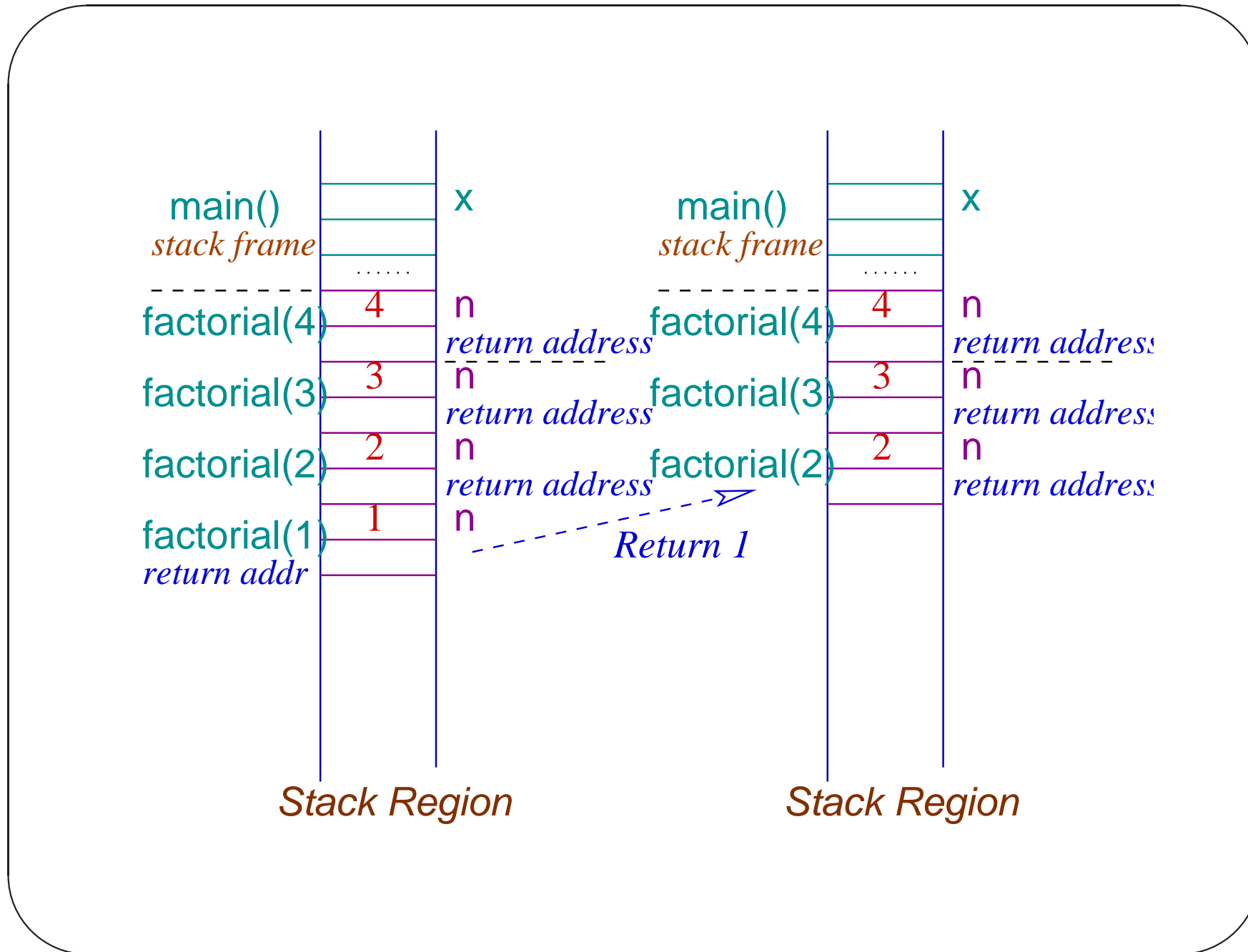
Note

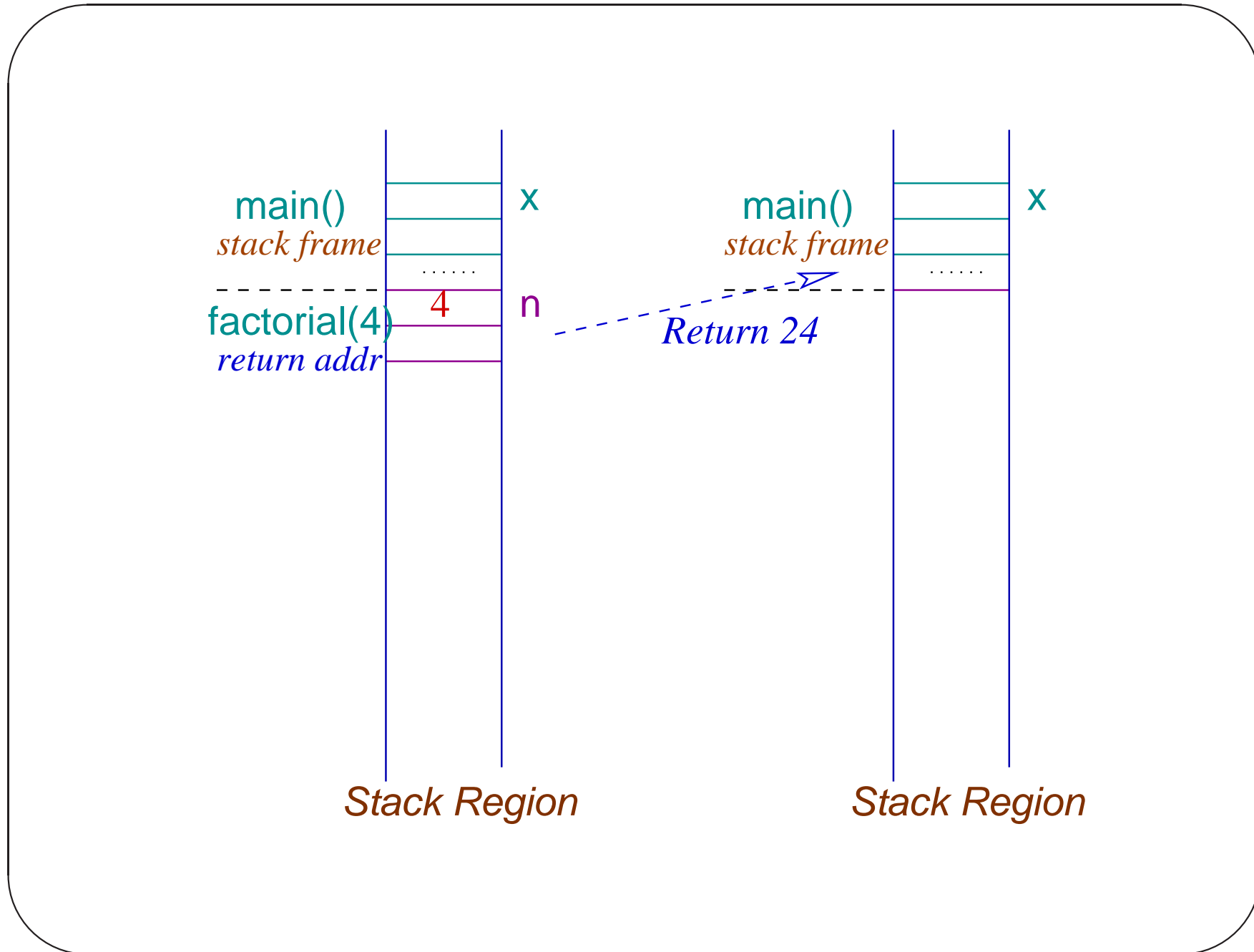
- Once a **return** statement is executed, all the variables of the corresponding invocation die.
- The last incarnation of a variable name dies first - **last in first out (LIFO)**. The **last call** is returned **first**.

Stack Frame or Activation Record









Note

- The recursive factorial function uses **more memory** than its non-recursive counter part.
- The non-recursive function uses **fixed amount** of memory for an `int` data, whereas the memory usage by the recursive function is **proportional** to the value of data.
- Moreover a function call and return takes some amount of extra time.

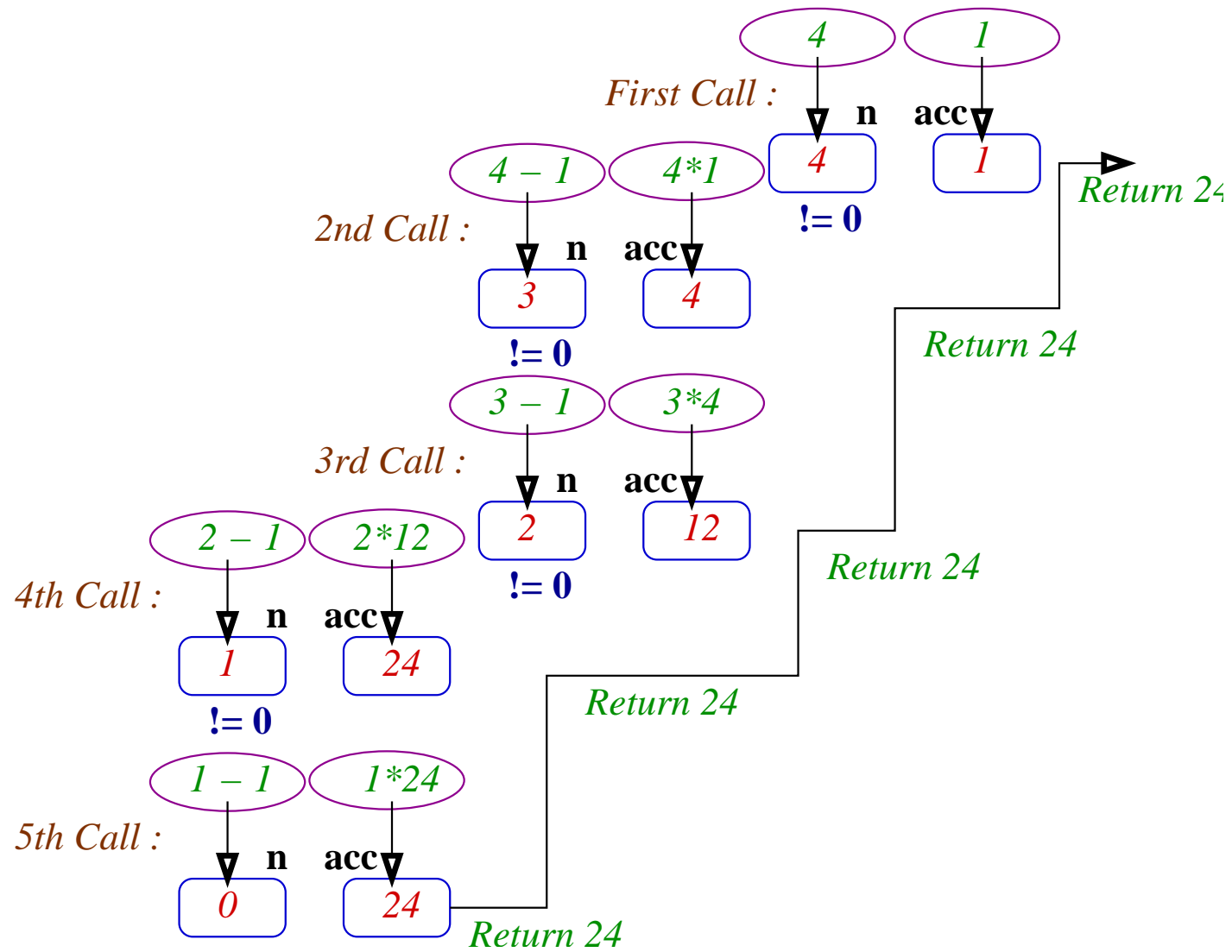
Recursive Function with Iterative Dynamics

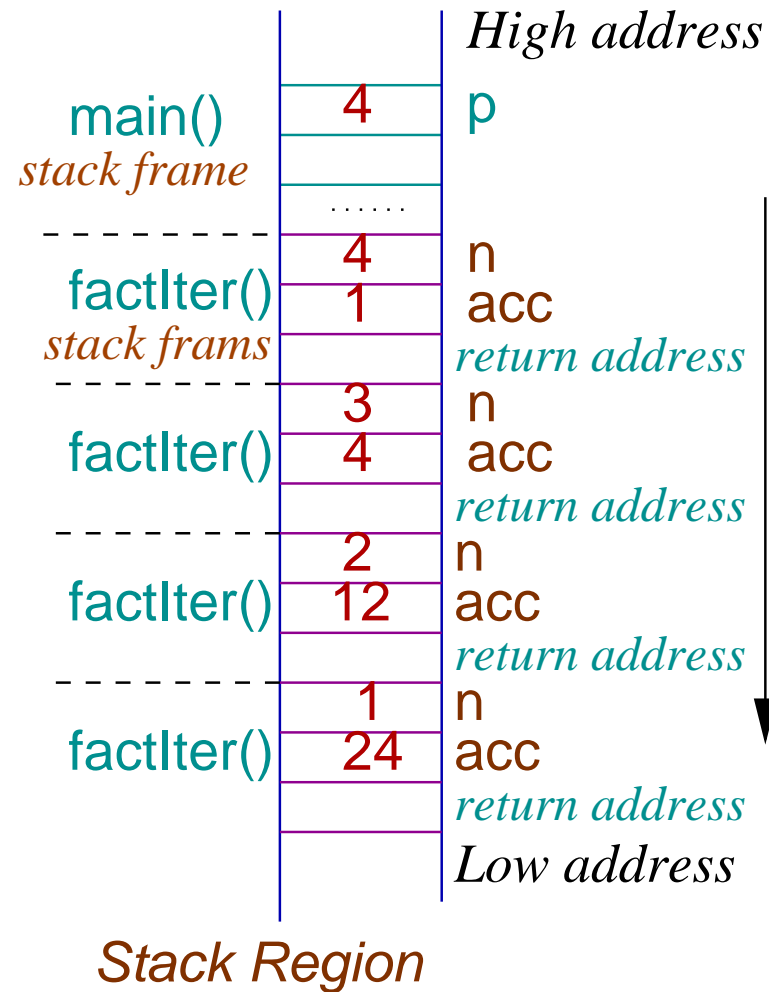
We have seen that the **value computation** in our factorial function starts after unfolding the recursion. But this dynamics of computation in a recursive function can be made different. The function may start the computation from the beginning.

```
int factIter(int n, int acc)
{
    if (n == 0) return acc ;
    else return factIter(n-1, n*acc) ;
} // factorialFR2.c
```

This function is called as `factIter(n, 1)` to calculate the value of $n!$. The second parameter is the value of the `basis`.

Computation of `factIter(4, 1)`





Note

The computation of $n!$ in this recursive function is very similar to the computation in a **for-loop**.

```
int factIter(int n) {  
    int acc = 1, i ;  
  
    for(i = n; i > 0 ; --i) acc *= i ;  
    return acc ;  
}
```

Inductive Definition: $\text{gcd}(m, n)$

$$\text{gcd}(m, n) = \begin{cases} n & \text{if } m = 0, \\ \text{gcd}(n \bmod m, m) & \text{if } m > 0. \end{cases}$$

Recursive Function gcd(m,n)

```
int gcd(int s, int l){  
    if(s == 0) return l;  
    return gcd(l%s, s);  
} // gcdFR.c
```

Different Calls to `gcd()`

`gcd(0, 0)` \Rightarrow return 0

`gcd(0, 5)` \Rightarrow return 5

`gcd(5, 0)` \Rightarrow `gcd(0, 5)`

\Rightarrow return 5

`gcd(18, 12)` \Rightarrow `gcd(12, 18)`

\Rightarrow `gcd(6, 12)`

\Rightarrow `gcd(0, 6)`

\Rightarrow return 6