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].

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

[a]The 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.

$$\texttt{A()} \xrightarrow{\texttt{call A}} \texttt{A()} \xrightarrow{\texttt{call A}} \texttt{A()} \xrightarrow{\texttt{call A}} \texttt{A()} \cdots$$

Recursive Call

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,

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

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

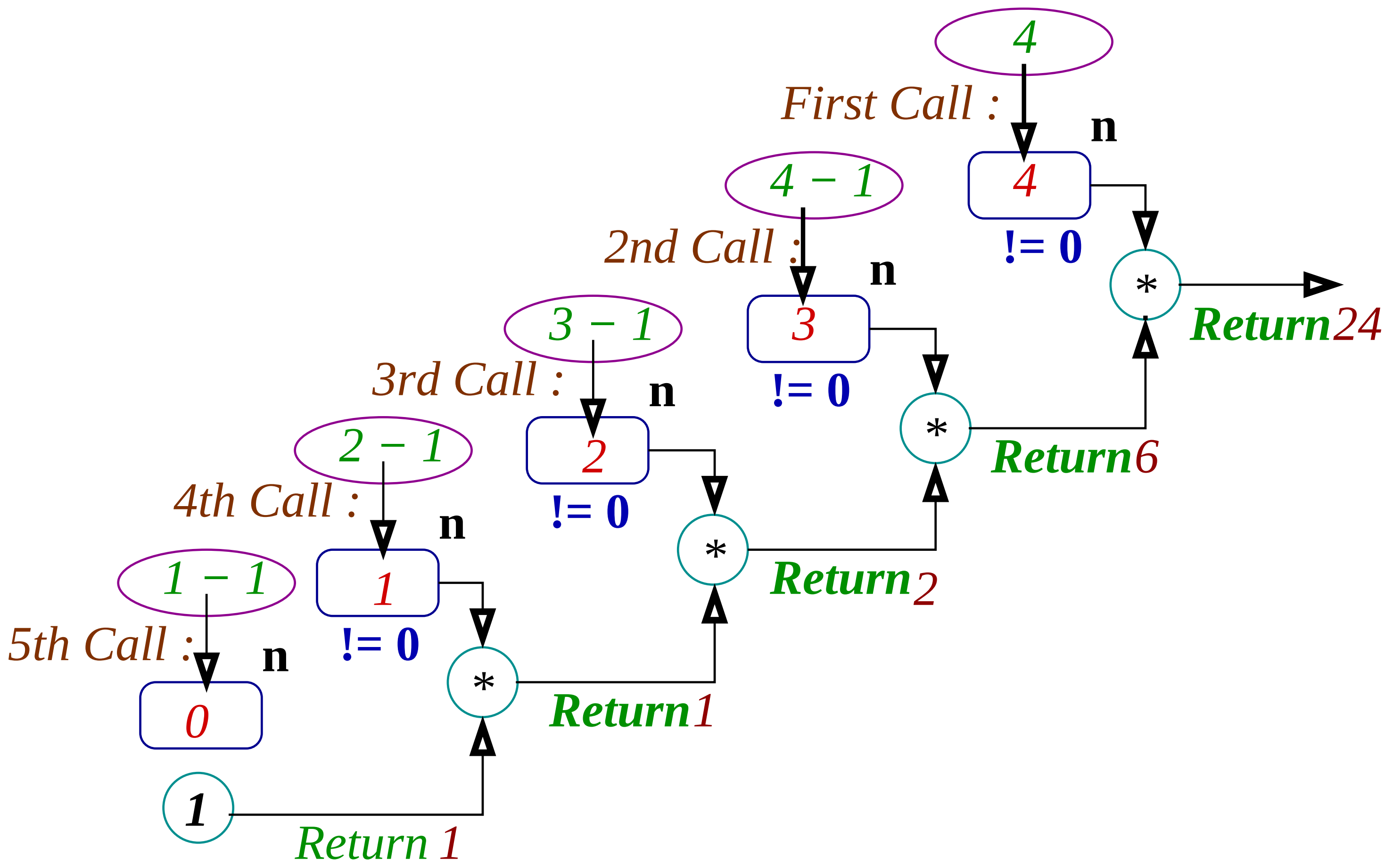
Recursive factorial Function

```
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 $p!$.

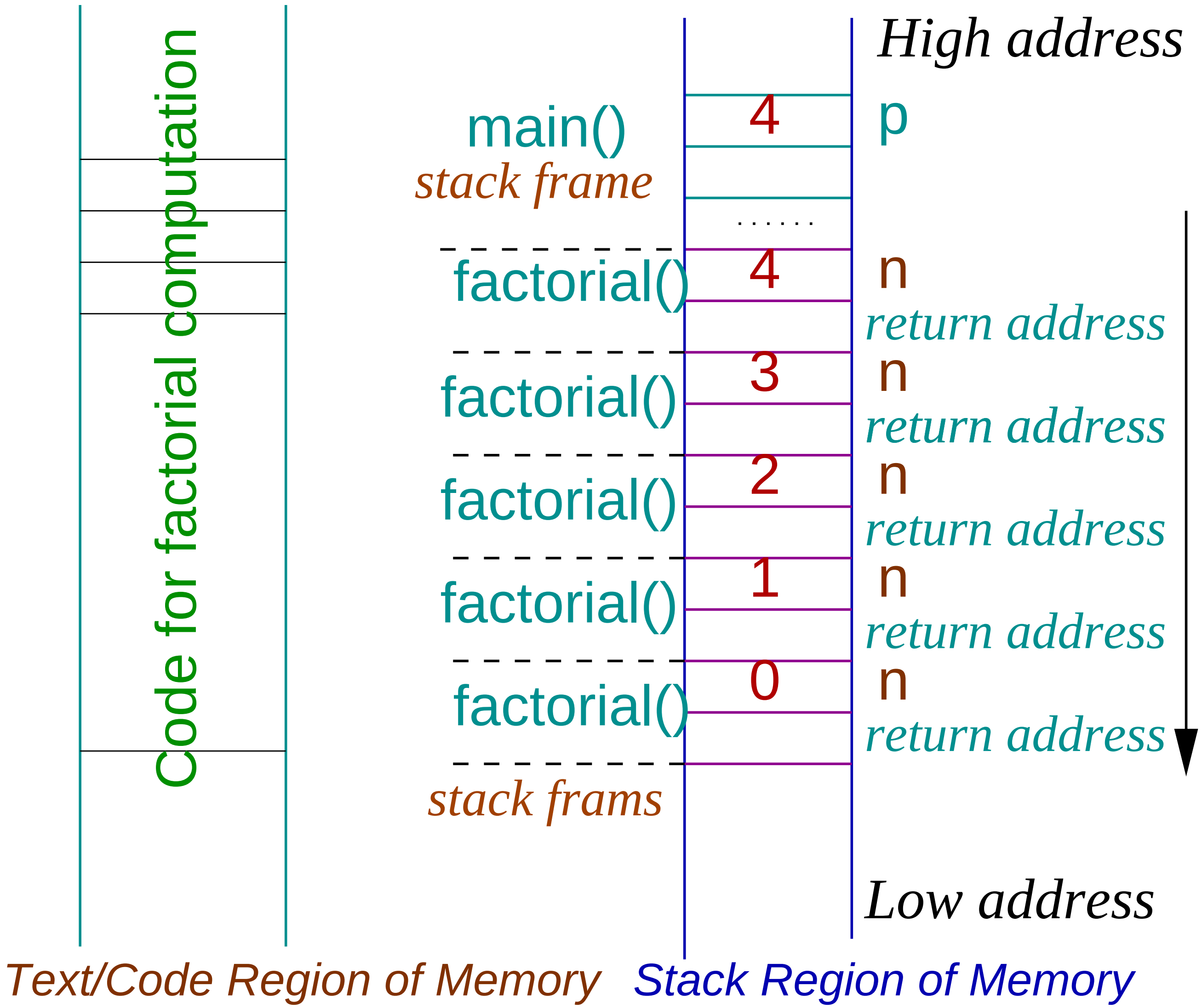
Different Calls and Incarnations of n

Actual Parameter is 4


4
First Call :
n
4
!= 0
*
Return24
4 − 1
2nd Call :
n
3
!= 0
*
Return6
3 − 1
3rd Call :
n
2
!= 0
*
Return2
2 − 1
4th Call :
n
1
!= 0
*
Return1
1 − 1
5th Call :
n
0
1
Return 1

Same Code but Different Data

- Same code is used for every recursive call.
- But data changes in every recursive call.


High address
Code for factorial computation
main()
stack frame
4
p
......
factorial()
4
n
return address
factorial()
3
n
return address
factorial()
2
n
return address
factorial()
1
n
return address
factorial()
0
n
return address
stack frams
Low address
Text/Code Region of Memory
Stack Region of Memory

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.

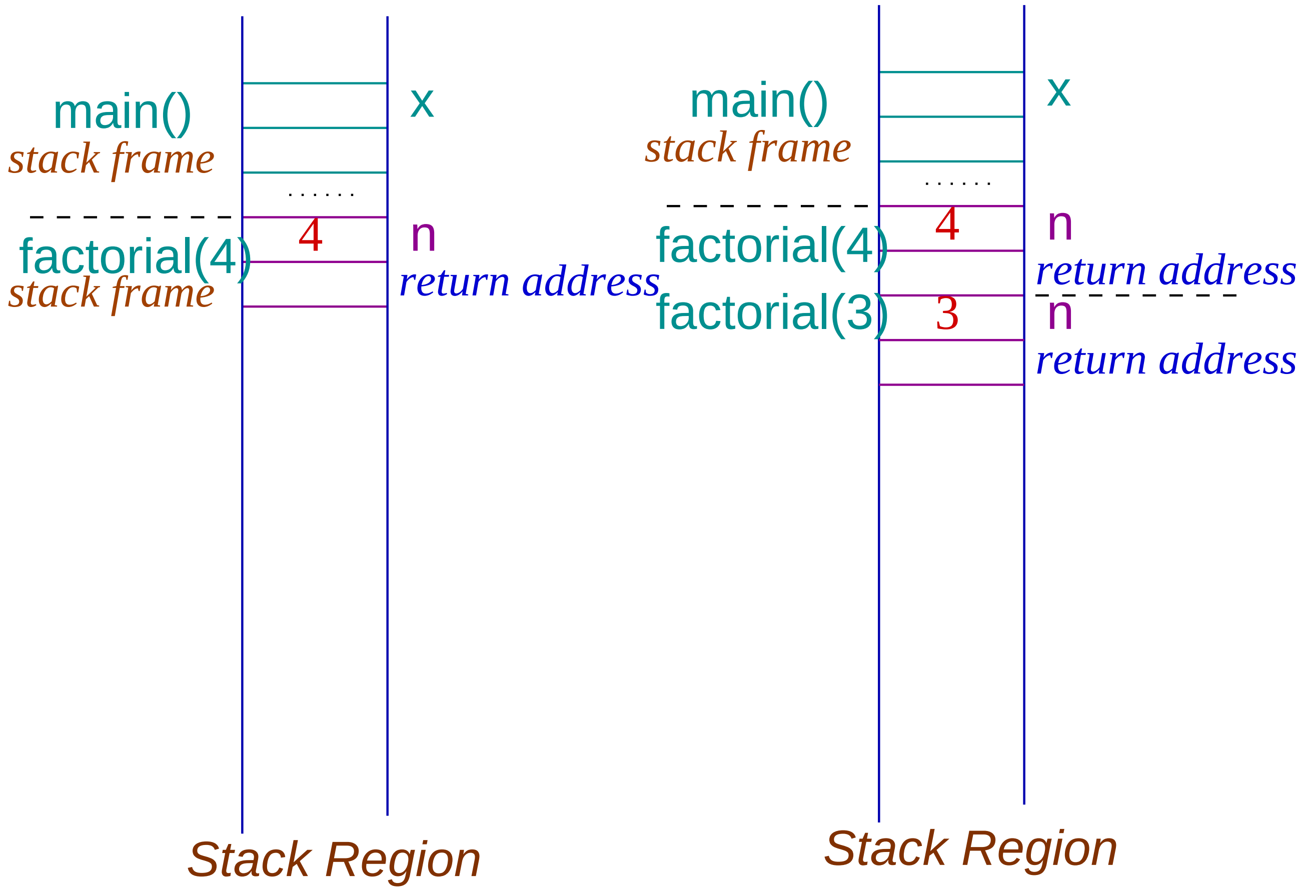
Note

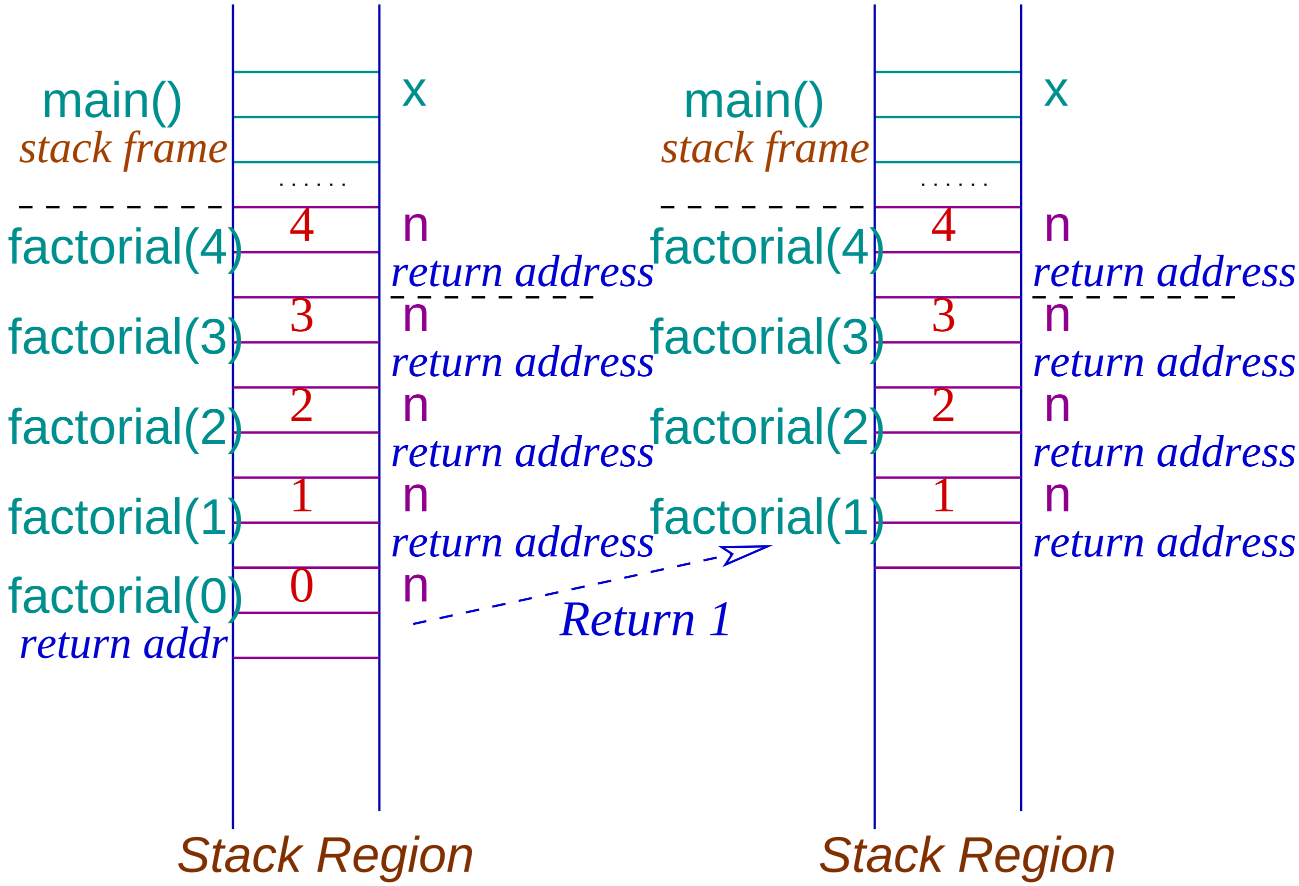
- For every call there is new incarnation of all the formal parameters and the local variables (that are not static). The variable names get bind to different memory locations.
- Variables of one invocation are not visible from another invocation.

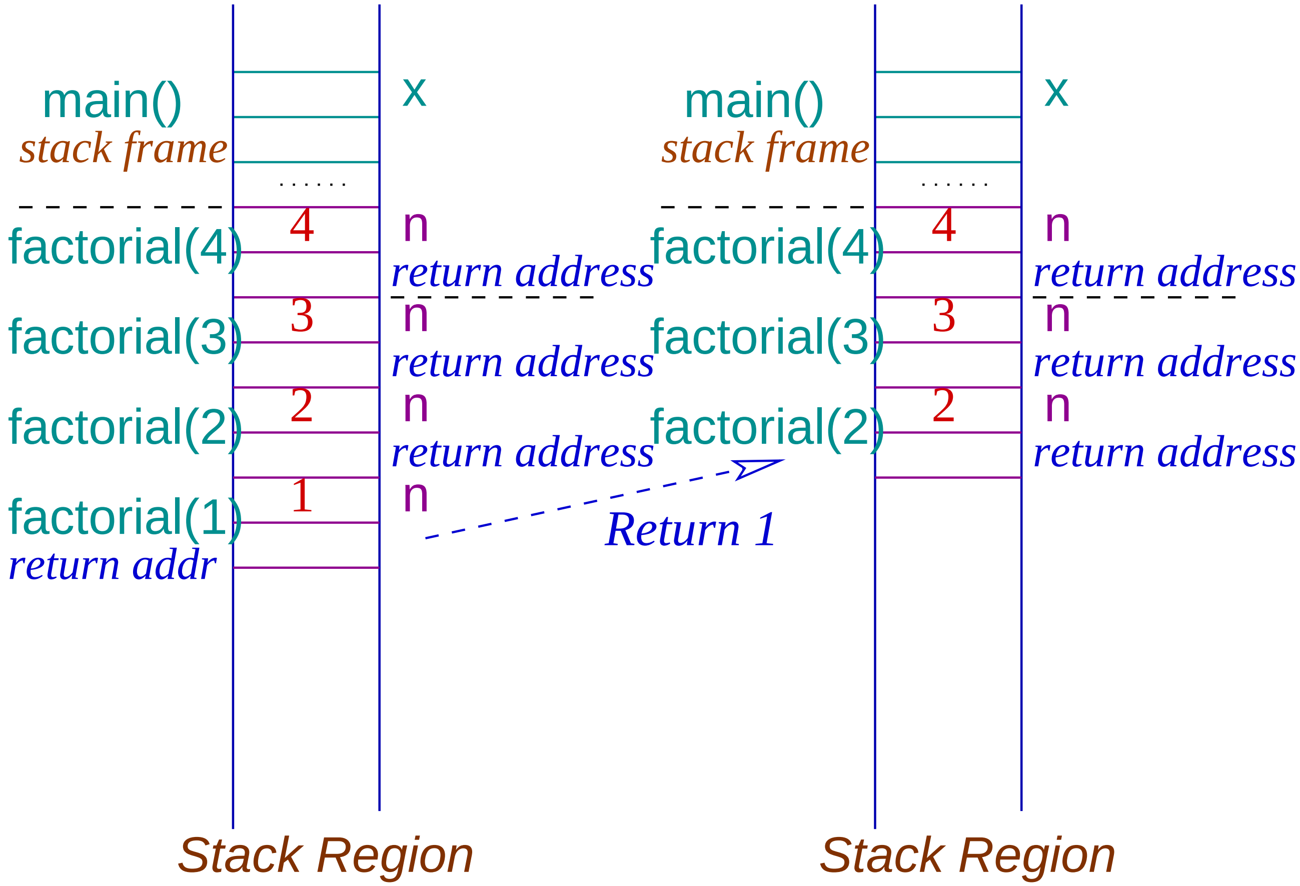
Note

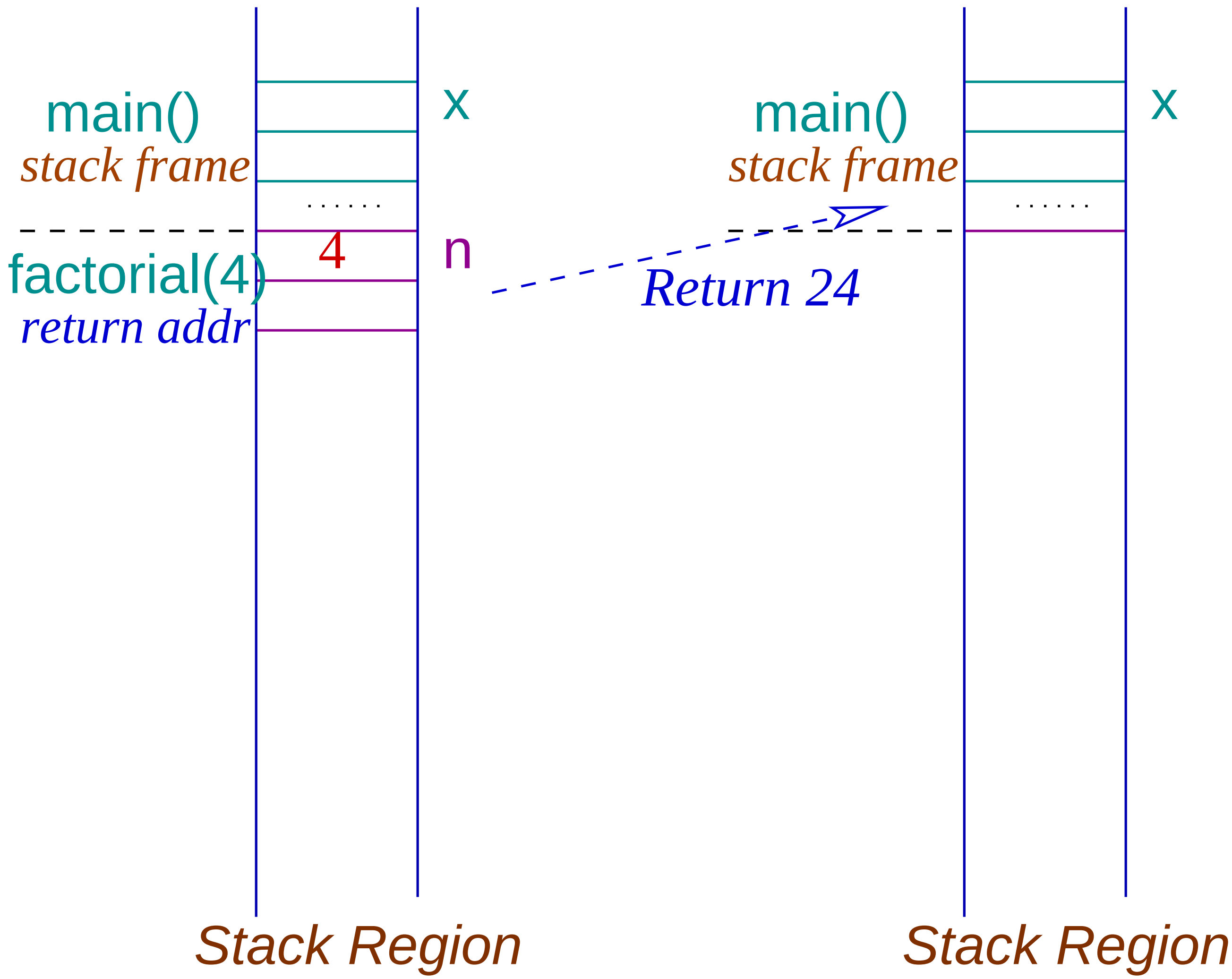
- Once a return statement is executed, all the variables of the corresponding invocation die.
- The last incarnation of a variable name dies first - last in first out (LIFO). The last call is returned first.

Stack Frame or Activation Record




main()
stack frame
x
......
factorial(4)
4
n
return address
factorial(3)
3
n
return address
factorial(2)
2
n
return address
factorial(1)
1
n
return address
factorial(0)
0
n
return addr
Return 1
Stack Region
main()
stack frame
x
......
factorial(4)
4
n
return address
factorial(3)
3
n
return address
factorial(2)
2
n
return address
factorial(1)
1
n
return address
Stack Region


main()
stack frame
x
......
factorial(4)
4
n
return address
factorial(3)
3
n
return address
factorial(2)
2
n
return address
factorial(1)
1
n
return addr
Return 1
Stack Region
main()
stack frame
x
......
factorial(4)
4
n
return address
factorial(3)
3
n
return address
factorial(2)
2
n
return address
Stack Region


main()
stack frame
x
......
factorial(4)
4
n
return addr
Return 24
main()
stack frame
x
......
Stack Region
Stack Region

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.

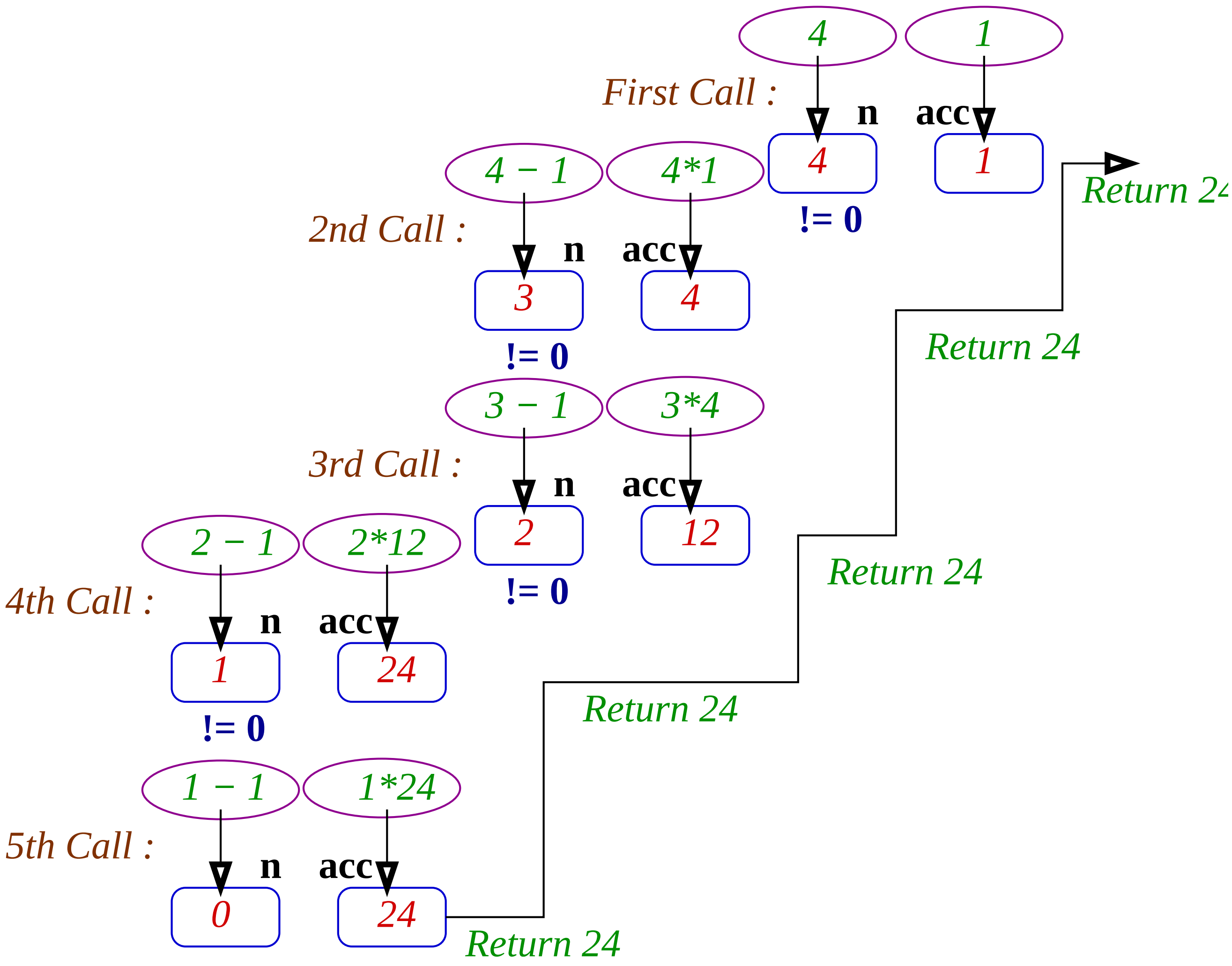
Recursive Function with Iterative Dynamics

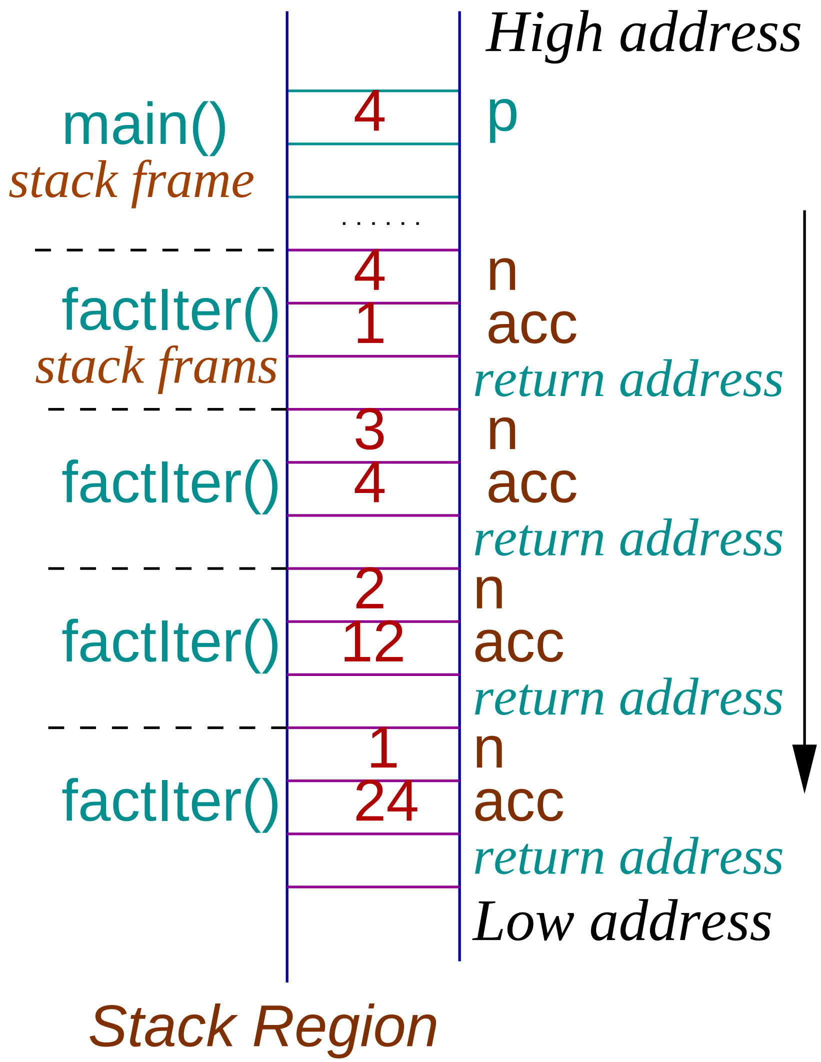
We have seen that the value computation in our factorial function starts after unfolding the recursion. But this dynamics of computation in a recursive function can be made different. The function may start the computation from the beginning.

```
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)`




High address
main()
stack frame
4
p
......
factIter()
stack frams
4
n
1
acc
return address
factIter()
3
n
4
acc
return address
factIter()
2
n
12
acc
return address
factIter()
1
n
24
acc
return address
Low address
Stack Region

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: $\gcd(m, n)$

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

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()

$$
\begin{aligned}
gcd(0,0) &\Rightarrow \text{return } 0\\
gcd(0,5) &\Rightarrow \text{return } 5\\
gcd(5,0) &\Rightarrow gcd(0,5)\\
&\Rightarrow \text{return } 5\\
gcd(18,12) &\Rightarrow gcd(12,18)\\
&\Rightarrow gcd(6,12)\\
&\Rightarrow gcd(0,6)\\
&\Rightarrow \text{return } 6
\end{aligned}
$$