Recursion

Luca Abeni
`luca.abeni@santannapisa.it`

Execution as Evaluation

- Functional program: composition of pure functions
  - Recursion is used instead of iteration
- “Executed” by evaluating the expressions obtained from the functions
- Usual example: factorial!

```
unsigned int fact(unsigned int n)
{
  return n == 0 ? 1: n * fact(n - 1);
}
```

  - Note the “arithmetic if” (p ? a : b)
- `fact(4)` = ?

Example of Evaluation

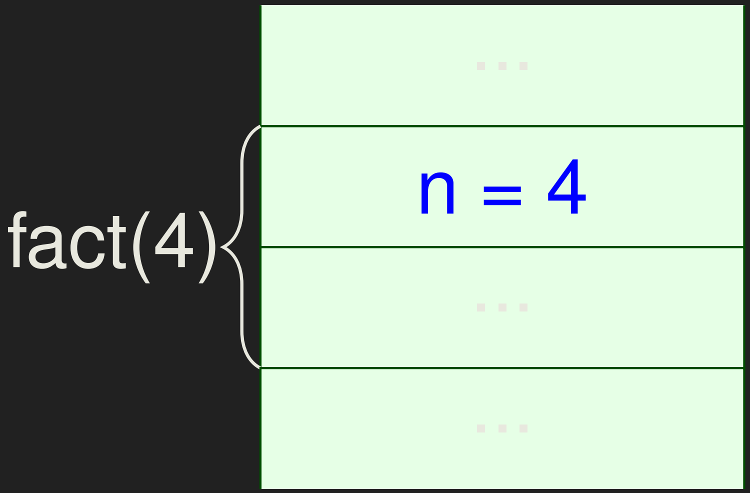
`fact(4) = ...` “n == 0 ? 1: n $*$ fact(n − 1)”,
replacing “`n`” with “`4`”

- (n == 0 ? 1: n $*$ fact(n − 1))(4)
- So, 2 different replacements: replace “`fact`” with its definition, and then replace “`n`” with “`4`”

```
fact(4) = (4 == 0) ? 1 : 4 * fact(3) =
4 * fact(3) =
4 * ((3 == 0) ? 1 : 3 * fact(2)) =
4 * 3 * fact(2) =
4 * 3 * ((2 == 0) ? 1 : 2 * fact(1)) =
4 * 3 * 2 * fact(1) =
4 * 3 * 2 * ((1==0) ? 1 : 1*fact(0)) =
4 * 3 * 2 * 1 * 1 = 24
```

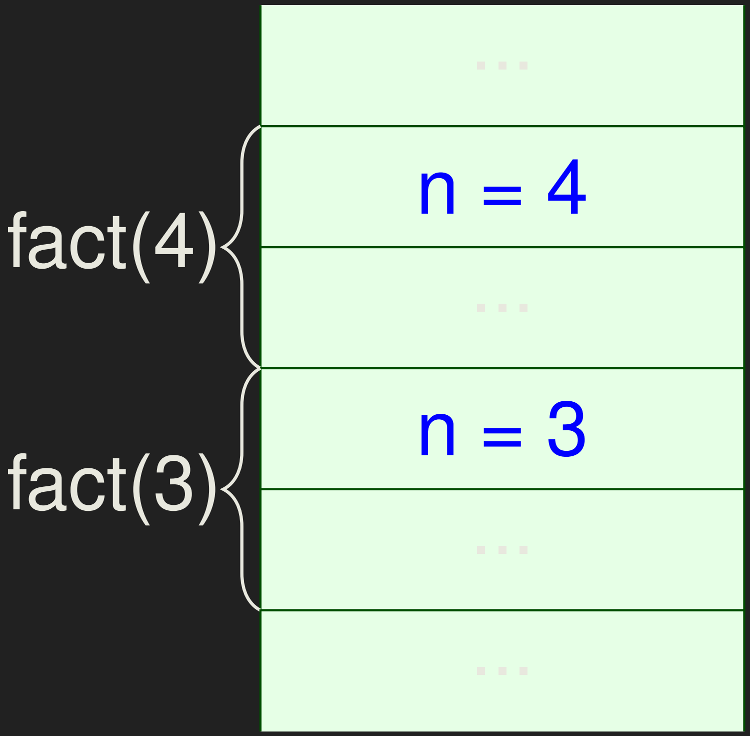
What About the Stack?

- Function invocation → activation record (stack frame) allocated on the stack...
- With recursion, this can be interesting!
- `fact(4)`: new stack frame containing:
  - The formal parameter $n = 4$
  - Link to previous stack frames
  - Some space for the return value



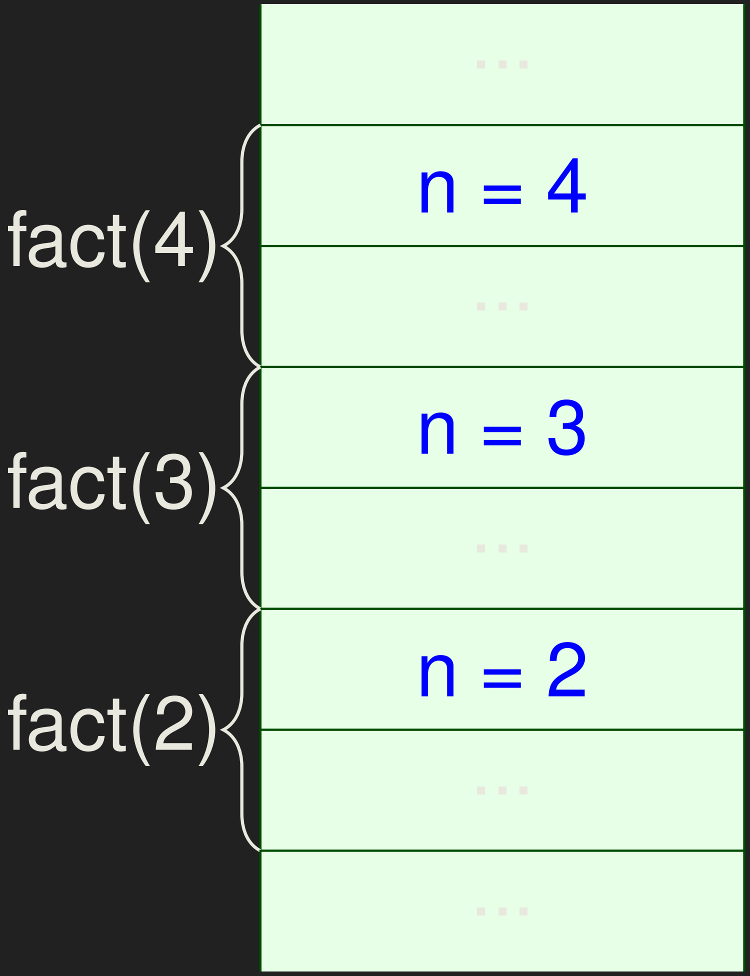
Stack Frame - 2

- `4 * fact(3)`



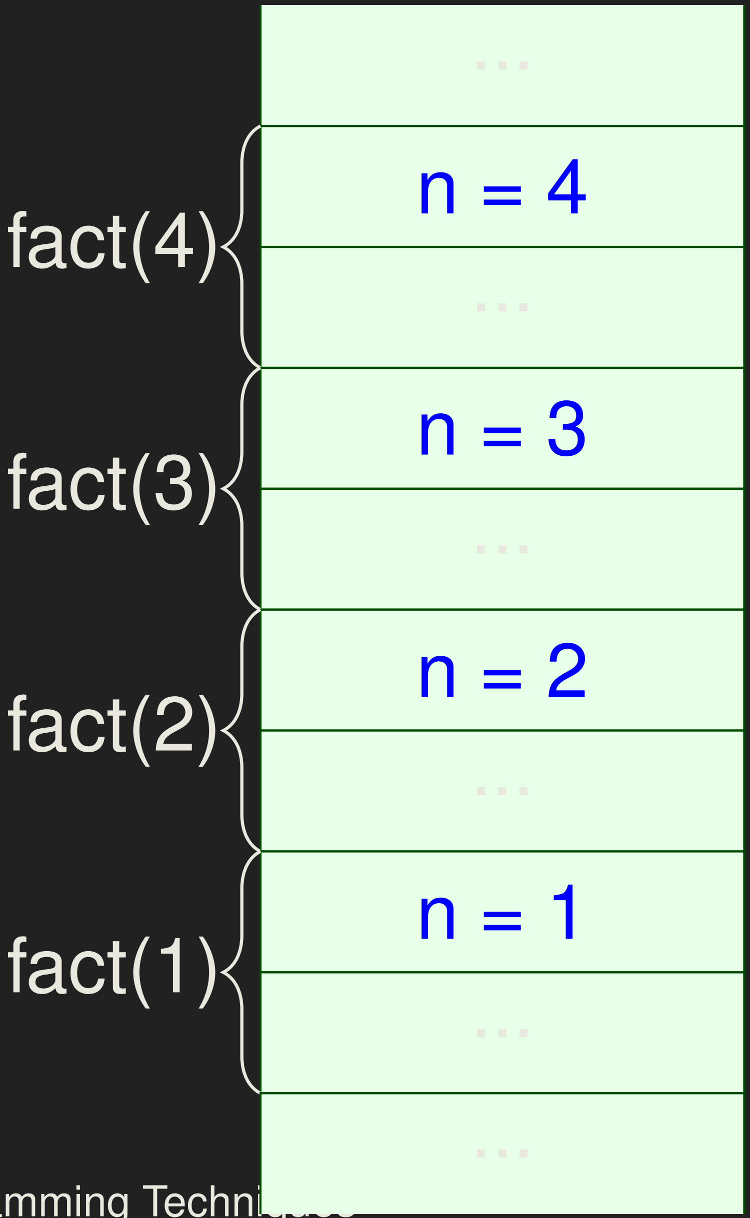
Stack Frame - 3

- `4 * 3 * fact(2)`



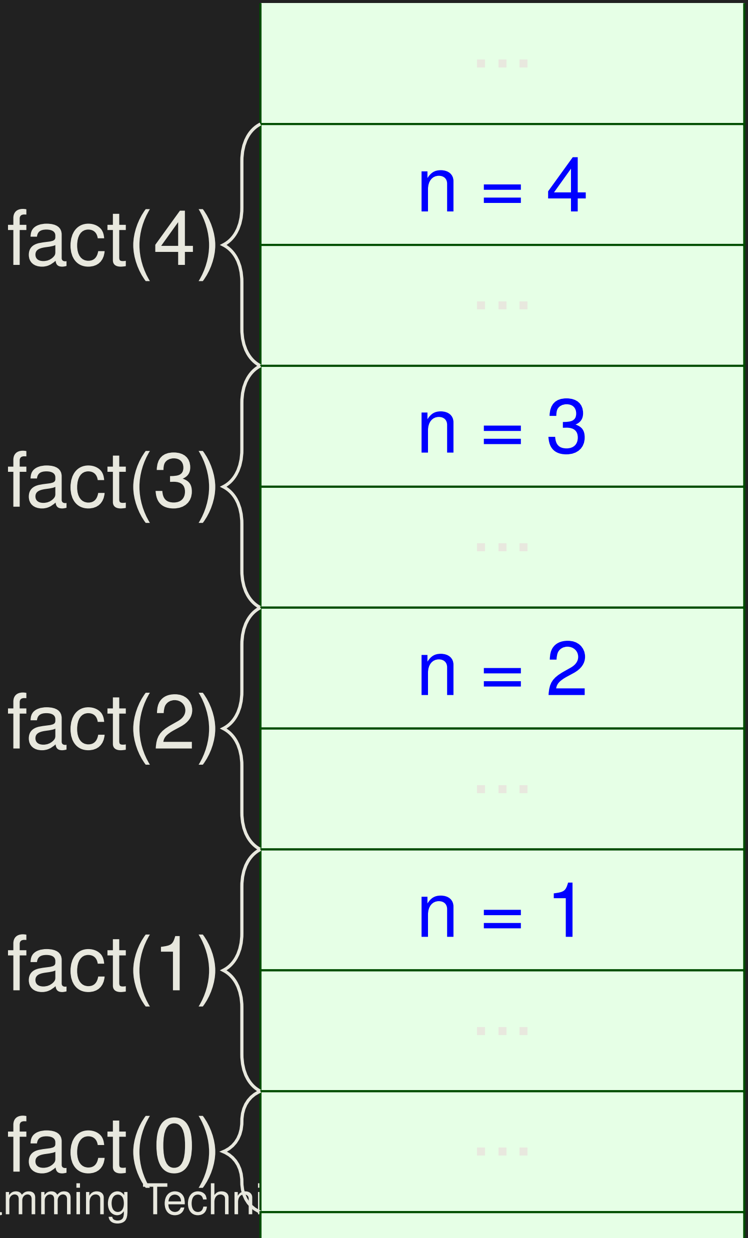
Stack Frame - 4

- `4 * 3 * 2 * fact(1)`



Stack Frame - 5

- `4 * 3 * 2 * 1 * fact(0)`



...

fact(4)
n = 4
...

fact(3)
n = 3
...

fact(2)
n = 2
...

fact(1)
n = 1
...

fact(0)
...

Summing Up...

- When `fact(0)` is evaluated, the previous stack frames contain the numbers to be multiplied...
- These stack frames are removed one after the other when the `fact()` instances return, and the multiplications are performed
- When `fact(n - 1)` returns, `fact(n)` still need to perform a multiplication by `n`
  - It cannot immediately return!
- The stack frames are hence needed until the corresponding `fact()` instance returns, and they cannot be removed from the stack before that
  - Recursion ⇒ high stack usage!
  - Possible stack overflow

Recursion and Stack Usage

- Is stack usage the price to be paid for using recursion?
- Let’s consider this factorial implementation:

```
unsigned int fact1(unsigned int n,
                   unsigned int res)
{
  return n == 0 ? res : fact1(n - 1, n * res);
}
unsigned int fact(unsigned int n)
{
  return fact1(n, 1);
}
```

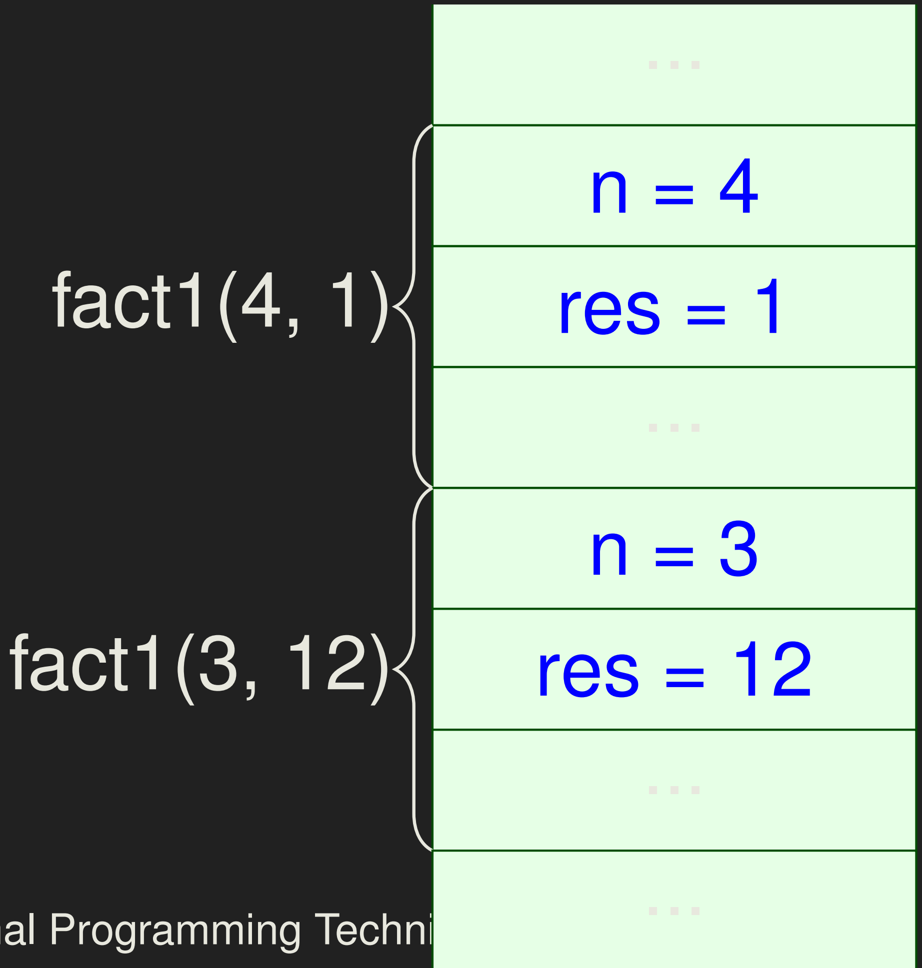
- What’s the second formal parameter???

Evaluation

```
fact(4) =
fact1(4, 1) =
(4 == 0) ? 1 : fact1(3, 4 * 1) =
fact1(3, 4) =
(3 == 0) ? 4 : fact1(2, 3 * 4) =
fact1(2, 12) =
(2 == 0) ? 12 : fact1(1, 2 * 12) =
fact1(1, 24) = fact1(0, 1 * 24) = 24
```

Stack Frames, Again

- No operations to be performed when `fact1(n-1, ...)` returns...
- The stack frame of `fact(n-1, ...)` already contains the data to return!



...

fact1(4, 1)

n = 4

res = 1

...

fact1(3, 12)

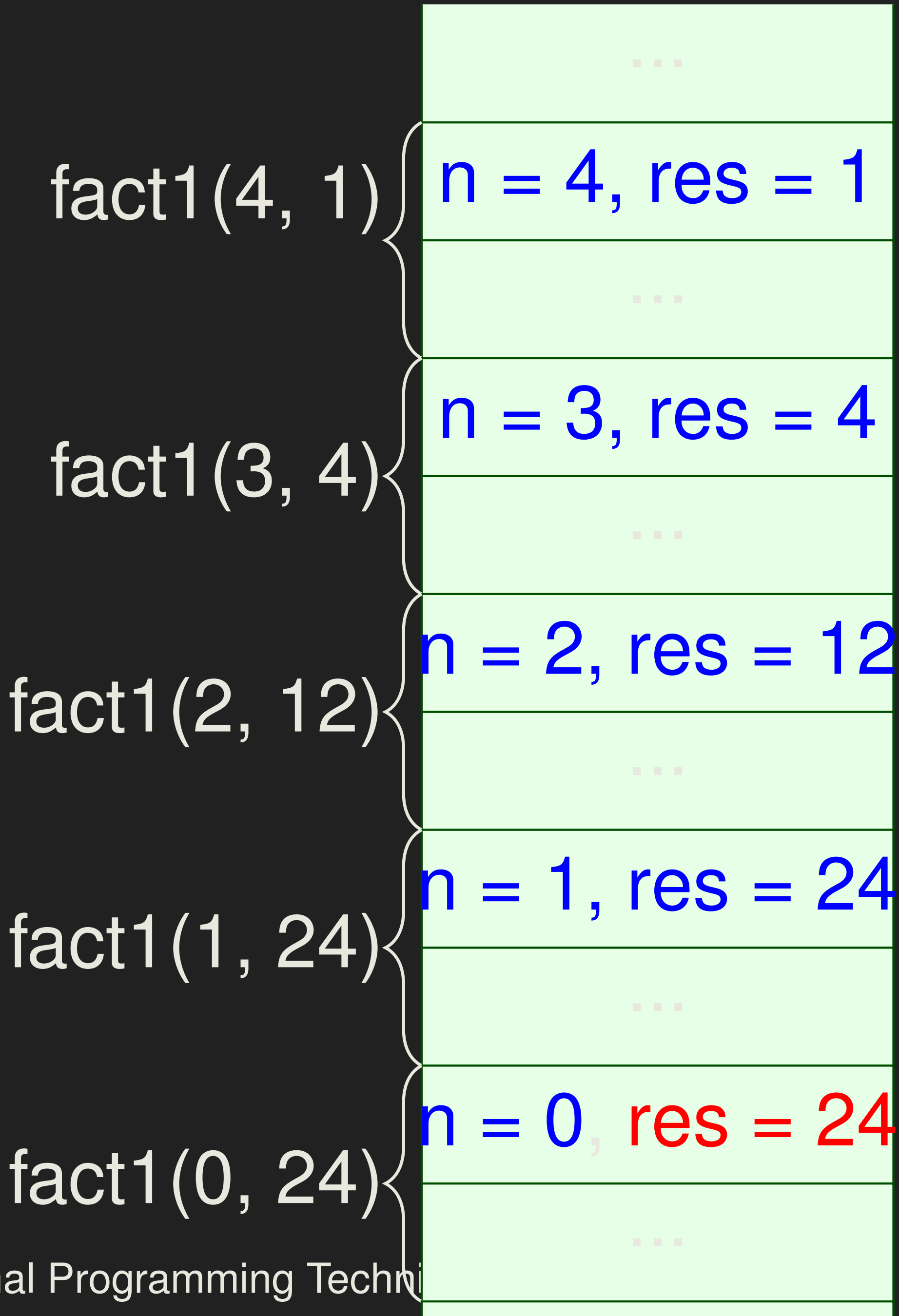
n = 3

res = 12

...

...

Stack Frames - 2



...

fact1(4, 1)

n = 4, res = 1

...

fact1(3, 4)

n = 3, res = 4

...

fact1(2, 12)

n = 2, res = 12

...

fact1(1, 24)

n = 1, res = 24

...

fact1(0, 24)

n = 0, res = 24

...

So...

- When `fact1(0, ...)` is evaluated, data from previous stack frames is not reused...
- Stack frames are removed when the `fact1()` instances return, without having to execute additional operations
- When `fact1(n - 1, ...)` returns, `fact1(n, ...)` returns its value directly
  - `fact1(n - 1, ...)` can immediately return to the `fact1(n, ...)` caller!
- Hence, stack frames can be removed from the stack when recursion is invoked (*before* the function returns)
  - Recursion ⇒ no additional stack usage
  - No stack overflow!