02393 Programming in C++

02393 Programming in C++
Module 10: Recursion

Lecturer:
Alceste Scalas

(Slides based on previous versions by Andrea Vandin, Alberto Lluch Lafuente, Sebastian Mödersheim)

3 November 2020

Lecture Plan

#	Date	Topic	Book chapter *
1	01.09	Introduction	
2	08.09	Basic C++	1
3	15.09	Data Types	2
4	22.09	Data Types / Libraries and Interfaces	2 / 3
5	29.09	Libraries and Interfaces	3
6	06.10	Classes and Objects	4.1, 4.2 and 9.1, 9.2
		Autumn break	
7	20.10	Templates	4.1, 11.1
8	27.10	LAB DAY	Old exams
9	03.11	Inheritance	14.3, 14.4, 14.5
10	10.11	Recursive Programming	5
11	17.11	Linked Lists	10.5
12	24.11	Trees	13
13	01.12	Summary & Exam Preparation	
	07.12	Exam	

* Recall that the book uses sometimes ad-hoc libraries that are slightly different with respect to the standard libraries (e.g., strings and vectors).

Recursion

What is Recursion?

- Solution technique that **solves large problems** by **reducing them to smaller problems of the same form**
- It is crucial that the smaller problem has the same form
- This means we can use the same technique for the big and the small problem!

Recursion

What is Recursion?

- Solution technique that **solves large problems** by **reducing them to smaller problems** **of the same form**
- It is crucial that the smaller problem has the same form
- This means we can use the same technique for the big and the small problem!

Why might recursion seem... weird?

- It requires a form of mathematical *inductive* reasoning
- Other programming concepts have a clearer real-life intuition
  - *loop*: repeat an action several times, on different objects
  - *if then else*: making decisions

Examples

Mathematical definitions often use recursion:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot ((n-1)!) & \text{otherwise} \end{cases}$$

Examples

Mathematical definitions often use recursion:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot ((n-1)!) & \text{otherwise} \end{cases}$$

A possible *iterative* implementation in C++:

```
int factorial = 1;
for (unsigned int i = n; i > 0; i--) {
    factorial = factorial * i;
}
```

Examples

Mathematical definitions often use recursion:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot ((n-1)!) & \text{otherwise} \end{cases}$$

A possible *iterative* implementation in C++:

```
int factorial = 1;
for (unsigned int i = n; i > 0; i--) {
    factorial = factorial * i;
}
```

A possible *recursive* implementation in C++:

```
int Fact(unsigned int n) {
    if (n == 0) return 1;
    else return n * Fact(n-1);
}
```

Examples

Mathematical definitions often use recursion:

$$Fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (Fact(n-1)) & \text{otherwise} \end{cases}$$

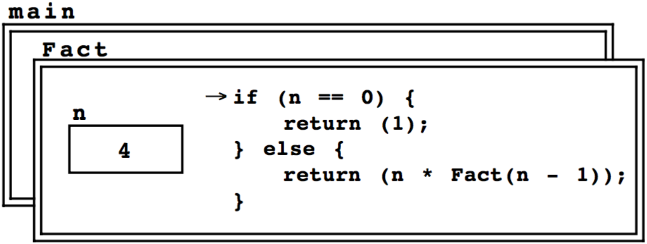
A possible *iterative* implementation in C++:

```
int factorial = 1;
for (unsigned int i = n; i > 0; i--) {
    factorial = factorial * i;
}
```

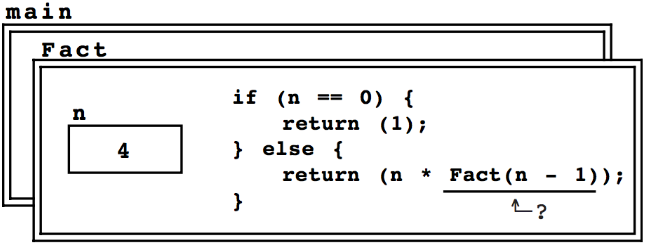
A possible *recursive* implementation in C++:

```
int Fact(unsigned int n) {
    if (n == 0) return 1;
    else return n * Fact(n-1);
}
```

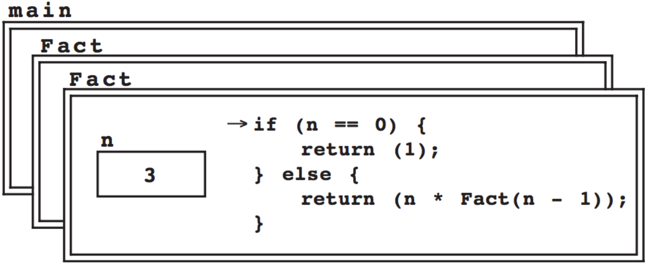
Executing the factorial



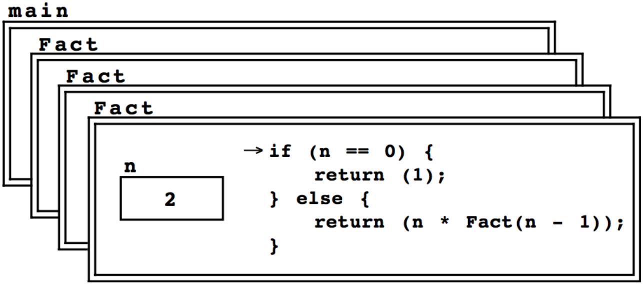
Executing the factorial (continued)



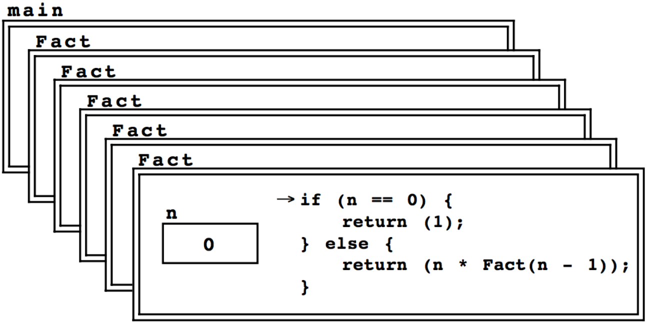
Executing the factorial (continued)



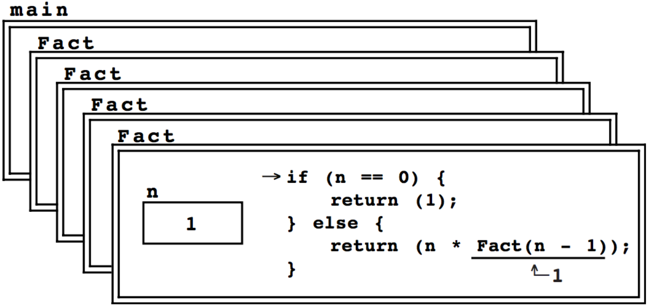
Executing the factorial (continued)



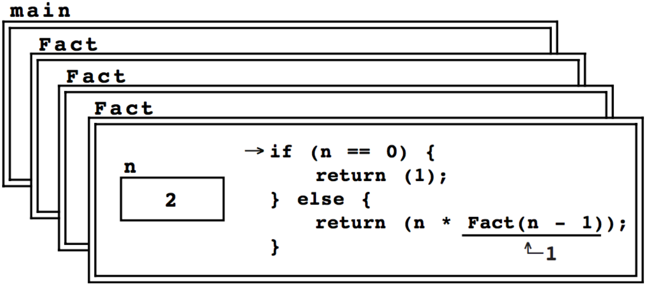
Executing the factorial (continued)



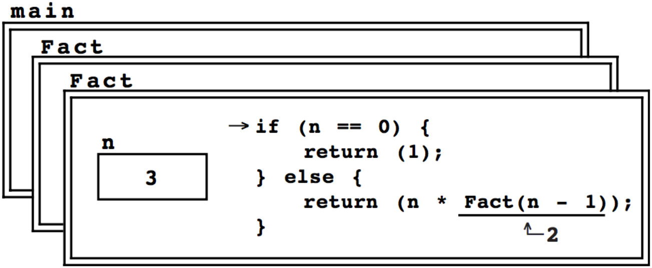
Executing the factorial (continued)



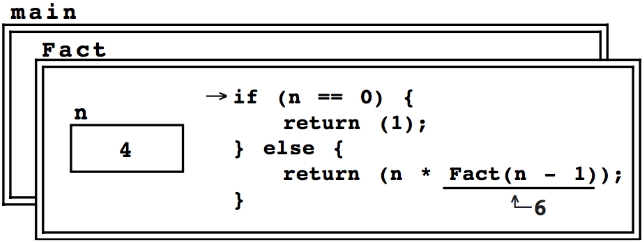
Executing the factorial (continued)



Executing the factorial (continued)



Executing the factorial (continued)



Recursion

When using recursion we must ensure:

1. Every recursion step reduces to a smaller problem
2. There is a smallest problem (or a set of smallest problems) that can be handled directly, without recursion
3. Every sequence of recursion steps eventually reaches one of such smallest problems

Recursion

When using recursion we must ensure:

1. Every recursion step reduces to a smaller problem
2. There is a smallest problem (or a set of smallest problems) that can be handled directly, without recursion
3. Every sequence of recursion steps eventually reaches one of such smallest problems

Otherwise?

- Risk of non-termination or crash (stack overflow)!

Recursive Leap of Faith

When writing a recursive function, we need to assume that each recursive call (with a smaller argument) computes the correct solution

- Example: to write `Fact(n)`, we need `Fact(n-1)` to be correct

Assuming that a recursive call works correctly is called the *Recursive Leap of Faith*

Recursion: Rules of Thumb

1. Identify the smallest cases before decomposition
2. Solve the smallest cases
3. Check that decomposition makes the problem simpler
4. Ensure that decomposition eventually reaches one of the smallest cases
5. Ensure that the arguments to the recursive calls are smaller versions of the original arguments
6. When you take the recursive leap of faith, ensure that recursive calls yield a correct solution to all smaller problems

Live coding

Another simple example: sum of n consecutive integers

On Complexity

Resources needed by an algorithm:

- **Time**: number of operations
- **Space**: amount of memory/disk

Resources depend on the size of the algorithm inputs (denoted N)

On Complexity

Resources needed by an algorithm:

- **Time**: number of operations
- **Space**: amount of memory/disk

Resources depend on the size of the algorithm inputs (denoted N)

Complexity of a problem:

- given a concrete problem (e.g., sorting a list of numbers)
- what is the best algorithm in terms of time and/or space?

On Complexity

Resources needed by an algorithm:

- **Time**: number of operations
- **Space**: amount of memory/disk

Resources depend on the size of the algorithm inputs (denoted N)

Complexity of a problem:

- given a concrete problem (e.g., sorting a list of numbers)
- what is the best algorithm in terms of time and/or space?

Notes:

- Some problems are not computable! (no algorithm exists)
- Sometimes trade-off between time and space
- For many problems, the precise complexity is not known:
  - ★ We have a best known algorithm
  - ★ We can give a lower bound

Asymptotic Complexity

Big-O notation and Big-Ω notation

We often focus on the worst-case time / space requirements of an algorithm, for input size N. We use **Big-O notation**. E.g.:

$$2N^2 + 17N + 53 \text{ operations} \quad \Longrightarrow \quad O(N^2) \text{ time complexity}$$

We only consider dominant terms because, as N grows,

- larger exponents have more impact
- constant factors and minor terms tend to become irrelevant

Asymptotic Complexity

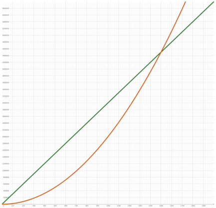
Big-O notation and Big-Ω notation

We often focus on the worst-case time / space requirements of an algorithm, for input size N. We use **Big-O notation**. E.g.:

$$2N^2 + 17N + 53 \text{ operations} \quad \Longrightarrow \quad O(N^2) \text{ time complexity}$$

We only consider dominant terms because, as N grows,

- larger exponents have more impact
- constant factors and minor terms tend to become irrelevant



For example, if we have:

1. a good algorithm, time: $3000N \implies O(N)$
2. a bad algorithm, time: $2N^2 \implies O(N^2)$

Above some N, algorithm (1) performs better

Definition (Big-O Notation)

$O(f)$: the class of functions that asymptotically grow no faster than f

$$O(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ : \exists N_0 \in \mathbb{N} : \forall N \geq N_0 : \ g(N) \leq c\, f(N)\}$$

For instance:

$$2N^2 + 17N + 53 < 73N^2$$

For $c = \frac{1}{73}$ and $N_0 = 1$, we obtain:

$$2N^2 + 17N + 53 \ \in \ O(N^2)$$

Dually, for giving lower-bounds on complexity, one uses $\Omega(f)$, which is the class of functions that grow at least as fast as f

More Examples of Recursion (see lecture code)

- Efficient search: binary search
  - Naive search (linear search) of an element in a set takes $O(n)$
  - Binary search is a divide-and-conquer $O(\log n)$ solution
- Efficient sorting: merge sort
  - The recursion paradigm directly triggers an efficient solution!
  - Naive bubble sort: $O(n^2)$ for array of size n
  - Merge sort: $O(n \log n)$ (theoretical optimum)
- Efficient exponentiation in cryptography ($a^n \ mod \ p$)
  - Naive exponentiation: $O(n)$
  - Efficient exponentiation: $O(\log n)$
  - Efficient solution is hard to program without recursion!

02393 Programming in C++