02393 Programming in C++

Module 10: Recursive Programming

Alceste Scalas <alcsc@dtu.dk>, **Giovanni Meroni** <giom@dtu.dk>

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

8 November 2022

Course plan

Module no.	Date	Topic	Book chapter*
0 and 1	30.08	Welcome & C++ Overview	1
2	06.09	Basic C++ and Data Types	2.1, 2.3 - 2.5, 11.1, 11.3
3	13.09	Enumerations and Structures & *LAB DAY*	1.5
4	20.09	Memory Management	12.1, 11.2, 11.3
5	27.09	Libraries and Interfaces	2.2, 2.6 - 2.8, 3.1 - 3.3, 4.1 - 4.3
6	04.10	Classes and Objects	5.1, 6.1, 11.2, 12.1, 12.4, 12.7
7	11.10	Templates	5.1, 14.2
		Autumn break	
8	25.10	Inheritance	19.1 - 19.3
9	01.11	*LAB DAY*	*Previous exams*
10	08.11	Recursive Programming	7
11	15.11	Linked Lists	12.2, 12.3
12	22.11	Trees	16.1 - 16.4
13	29.11	Conclusion & *LAB DAY*	*Exam preparation*
	22.12	**Exam (held physically, all aids allowed)**	

* Recall that the book uses some ad-hoc libraries (e.g., for vectors). We will use standard libraries

Outline

What is recursion?

Recursive programming is a solution technique that **solves large problems** by **reducing them to smaller problems of the same form**

- It is crucial that both the large and smaller problems have the same form!
- So we can use the same technique (and the same code!) to solve both

What is recursion?

Recursive programming is a solution technique that **solves large problems** by **reducing them to smaller problems of the same form**

- It is crucial that both the large and smaller problems have the same form!
- So we can use the same technique (and the same code!) to solve both

Why might recursion seem... weird?

- Recursion requires a form of mathematical **inductive reasoning**
- Recursion is **more abstract** than programming concepts having a mechanical intuition
  - *loop*: repeat an action several times
  - *if-then-else*: check a condition, make a decision
  - *recursion*: ???

Examples

You may have already seen recursion in **mathematical definitions**. E.g., the factorial:

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

Examples

You may have already seen recursion in **mathematical definitions**. E.g., the factorial:

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

A possible **iterative implementation** of the factorial in C++:

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

Examples

You may have already seen recursion in **mathematical definitions**. E.g., the factorial:

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

A possible **iterative implementation** of the factorial in C++:

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

A possible **recursive implementation** in C++:

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

Examples

You may have already seen recursion in **mathematical definitions**. E.g., the factorial:

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

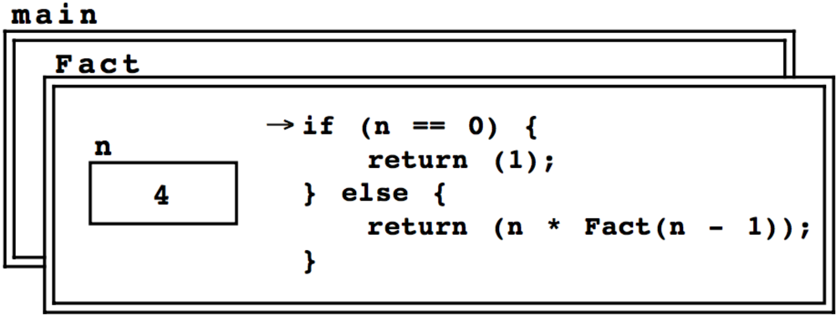
A possible **iterative implementation** of the factorial in C++:

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

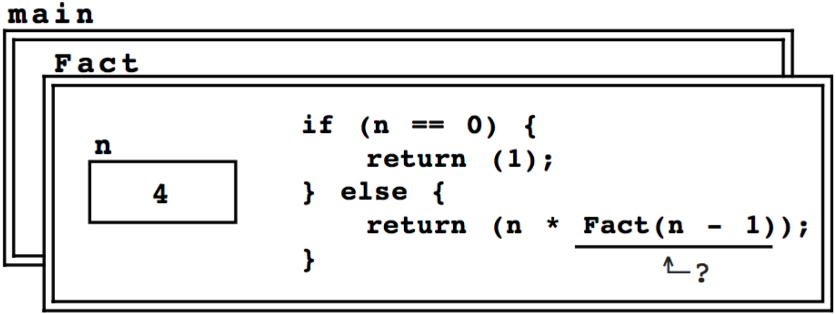
A possible **recursive implementation** in C++:

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

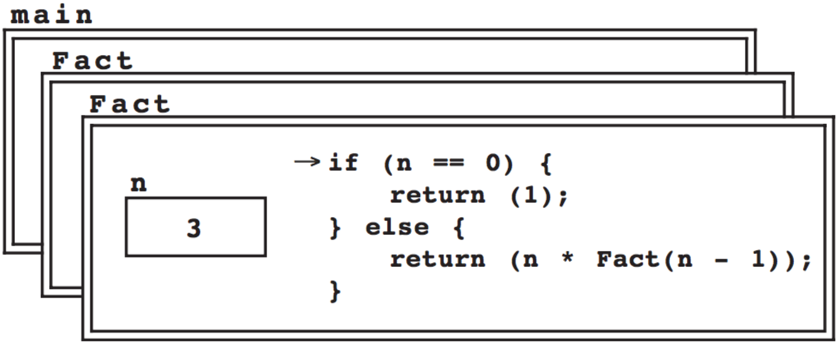
Example: executing `Fact(4)`



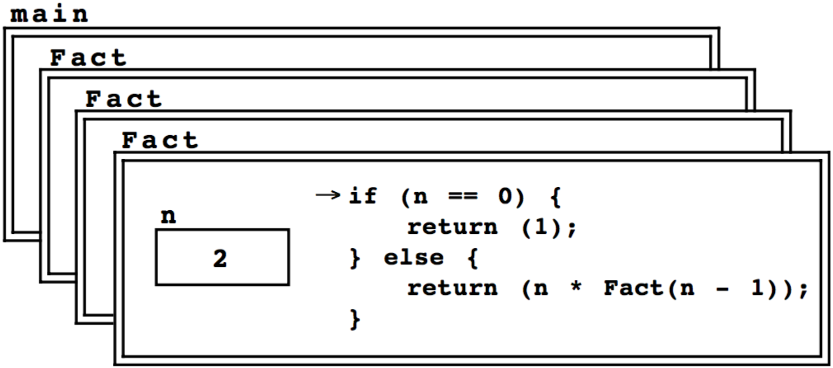
Example: executing `Fact(4)`



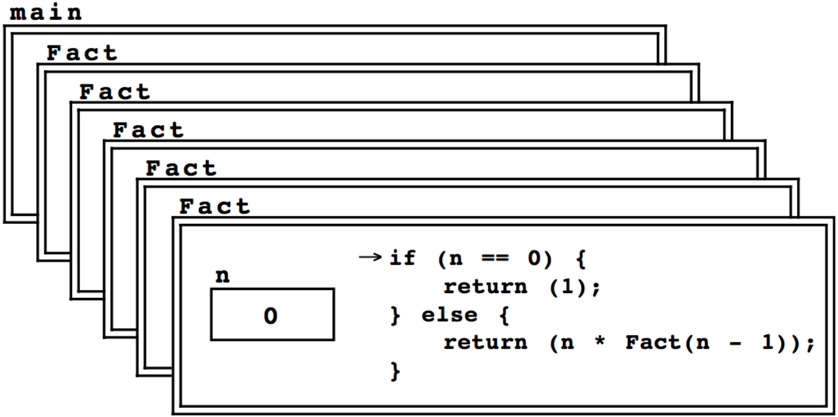
Example: executing `Fact(4)`



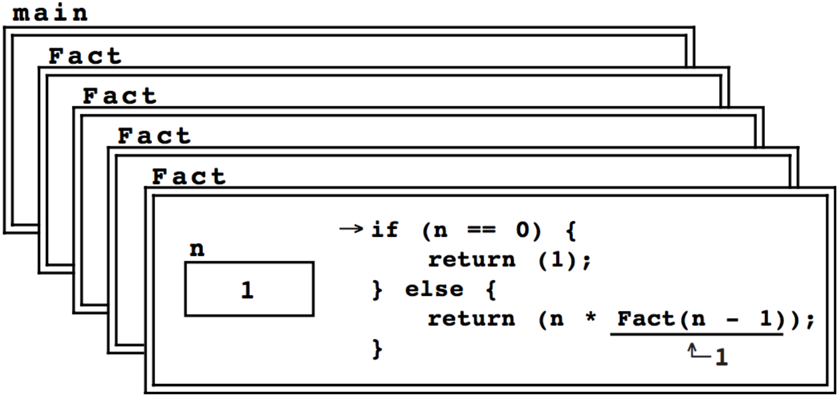
Example: executing `Fact(4)`



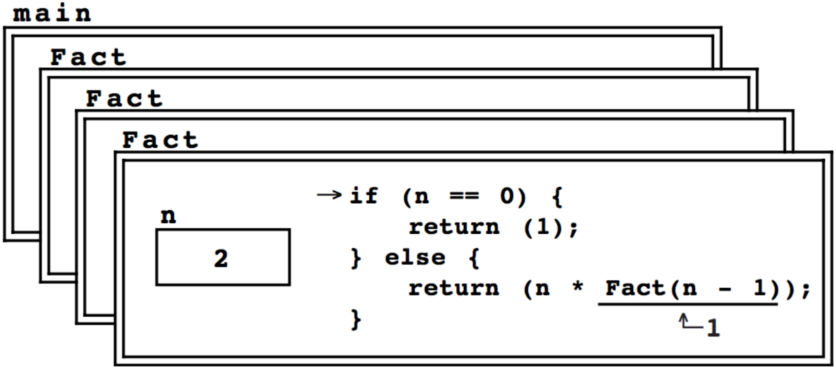
Example: executing `Fact(4)`



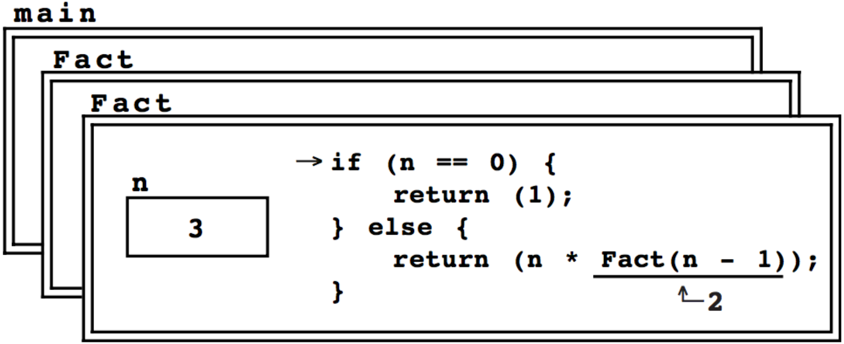
Example: executing `Fact(4)`



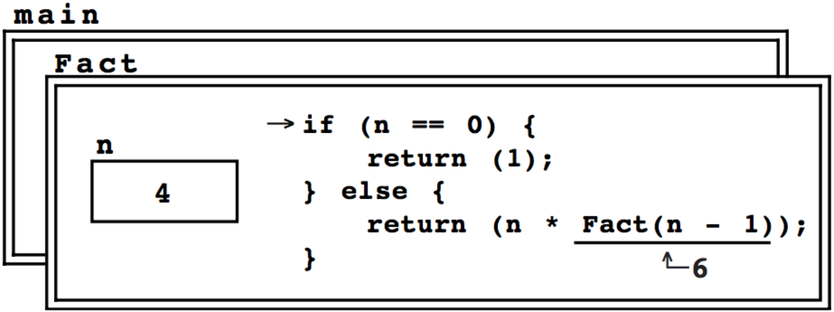
Example: executing `Fact(4)`



Example: executing `Fact(4)`



Example: executing `Fact(4)`



Using recursion: base cases and termination

When using recursion we must ensure that:

1. there are one or more **base cases**
   - i.e., **“smallest” problems** for which we return a solution, without recursion
2. every recursion step reduces to a **smaller problem**
3. every sequence of recursion steps **eventually reaches one of the base cases**

Using recursion: base cases and termination

When using recursion we must ensure that:

1. there are one or more **base cases**
   - i.e., **“smallest” problems** for which we return a solution, without recursion
2. every recursion step reduces to a **smaller problem**
3. every sequence of recursion steps **eventually reaches one of the base cases**

In the `Fact`orial example: the **base case** is `n = 0`, we recursively call `Fact` on **smaller and smaller numbers**, so we **eventually reach the base case** `n = 0`

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

Using recursion: base cases and termination

When using recursion we must ensure that:

1. there are one or more **base cases**
   - i.e., **"smallest" problems** for which we return a solution, without recursion
2. every recursion step reduces to a **smaller problem**
3. every sequence of recursion steps **eventually reaches one of the base cases**

In the `Fact`orial example: the **base case** is `n = 0`, we recursively call `Fact` on **smaller and smaller numbers**, so we **eventually reach the base case** `n = 0`

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

If any of the 3 conditions above is not satisfied, **recursion may not work properly!**

- Risk of **non-termination** or **crashes** (stack overflow)

Using recursion: the "recursive leap of faith"

When writing a recursive function, we need to **assume that each recursive call with a smaller problem computes the correct solution**

- Example: to write `Fact(n)`, we need to assume that `Fact(n-1)` is correct

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

Assuming that a recursive call works correctly is called the "**recursive leap of faith**"

Recursion: rules of thumb

1. **Identify the base cases** (i.e., the "smallest" cases whose solution does not need recursion)
2. **Solve the base cases**
3. **Address the recursive cases** (i.e., the non-base cases that need recursive calls)
   - Ensure that the arguments to the recursive calls are "smaller" than the original arguments
   - Ensure that recursive calls eventually reach one of the base cases

Live coding

Another simple example:

sum of n consecutive integers

On the complexity of algorithms and problems

We are often interested in estimating the **resources needed by an algorithm**:

- **time**: number of operations
- **space**: amount of memory/disk

Such resources usually depend on the **size of the algorithm inputs (denoted N)**

On the complexity of algorithms and problems

We are often interested in estimating the **resources needed by an algorithm**:

- **time**: number of operations
- **space**: amount of memory/disk

Such resources usually 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 time/space resources are needed by the **best** solution algorithm?

On the complexity of algorithms and problems

We are often interested in estimating the **resources needed by an algorithm**:

- **time**: number of operations
- **space**: amount of memory/disk

Such resources usually 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 time/space resources are needed by the **best** solution algorithm?

Notes:

- Some problems are **not computable**! (i.e., no algorithm exists)
- Sometimes we have a **trade-off** between time and space
- For many problems, the **precise complexity is not known**:
  - we may know some solution algorithms, but we don't know whether a better one exists
  - we can give a **lower bound** to the complexity

Asymptotic complexity: Big-O notation

We are often interested in the **worst-case time / space** requirements of an algorithm, given an input size N. In such cases, we use **Big-O notation**. E.g.:

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

We **only consider dominant terms**. This is because, as N grows,

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

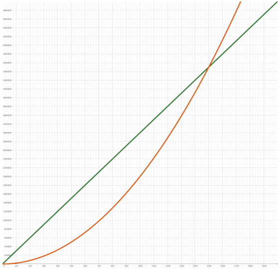
Asymptotic complexity: Big-O notation

We are often interested in the **worst-case time / space** requirements of an algorithm, given an input size N. In such cases, we use **Big-O notation**. E.g.:

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

We **only consider dominant terms**. This is 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: $3000N \implies O(N)$ time complexity
2. a **bad algorithm**, time: $2N^2 \implies O(N^2)$ time complexity

The plot shows N (x-axis) vs. number of operations (y-axis)
Above some N, the **algorithm (1)** performs better (less operations)

Asymptotic complexity: Big-O (cont'd)

Definition (Big-O notation). $O(f)$ is the class of functions that **asymptotically grow no faster** than f. More formally:

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

For instance, we have: $2N^2 + 2N + 1 \;\in\; O(N^2)$

This is because taking $c = 5$ and $N_0 = 1$, we have $2N^2 + 2N + 1 \;\leq\; cN^2$ for all $N \geq N_0$

Asymptotic complexity: Big-O (cont'd) and Big-Ω notation

Definition (Big-O notation). $O(f)$ is the class of functions that **asymptotically grow no faster** than f. More formally:

$$O(f) \;=\; \left\{\, g : \mathbb{N} \rightarrow \mathbb{R}^+ \;\middle|\; \exists c \in \mathbb{R}^+ : \exists N_0 \in \mathbb{N} : \forall N \geq N_0 : g(N) \leq c\, f(N) \,\right\}$$

For instance, we have: $2N^2 + 2N + 1 \;\in\; O(N^2)$

This is because taking $c = 5$ and $N_0 = 1$, we have $2N^2 + 2N + 1 \;\leq\; cN^2$ for all $N \geq N_0$

Dually, we are sometimes interested in the **best-case time / space** requirements (i.e., **lower-bound complexity**) of an algorithm, given an input size N

For this we use $\Omega(f)$ which is the class of functions that grow **at least as fast** as f — and the corresponding **Big-Ω notation**

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!

Lab

Today's lab begins now. Tasks:

- make sure C++ works on your computer, request help if it doesn't
- begin working on **Assignment 9**
- ask questions if something is unclear (including previous assignments)