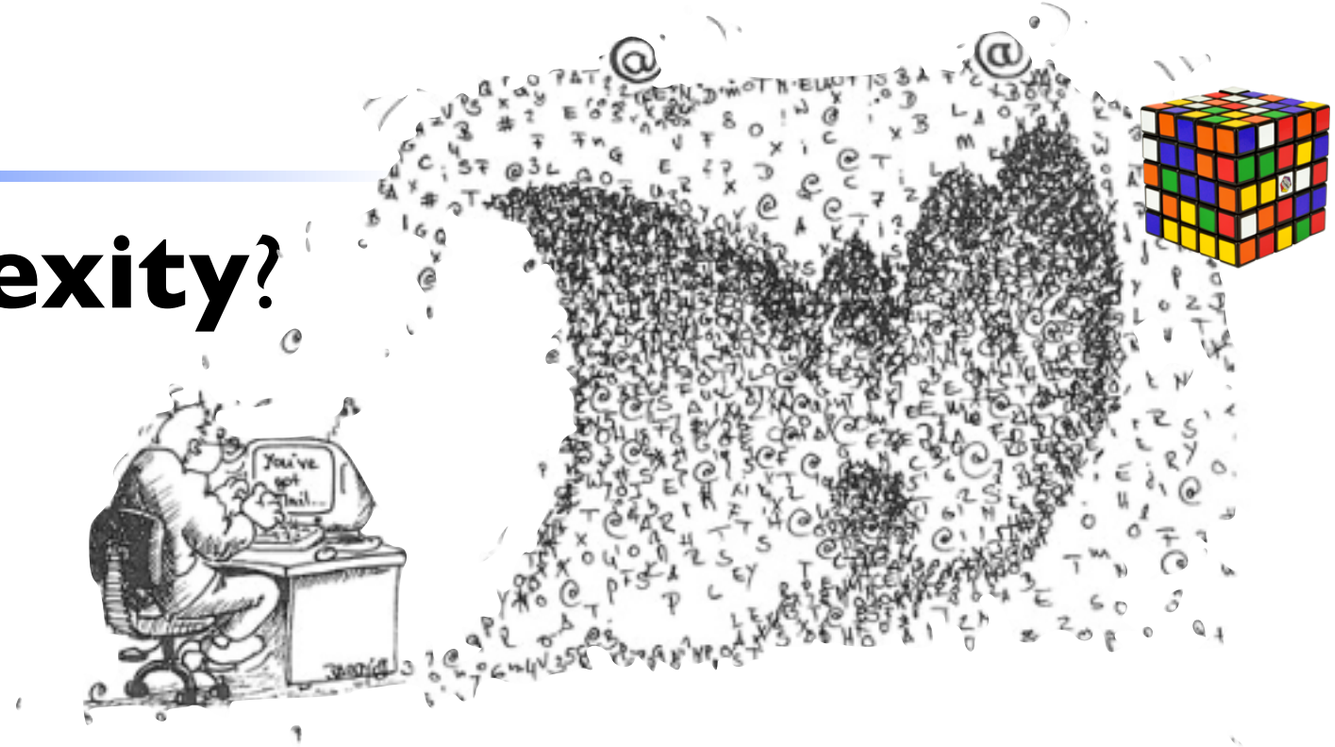
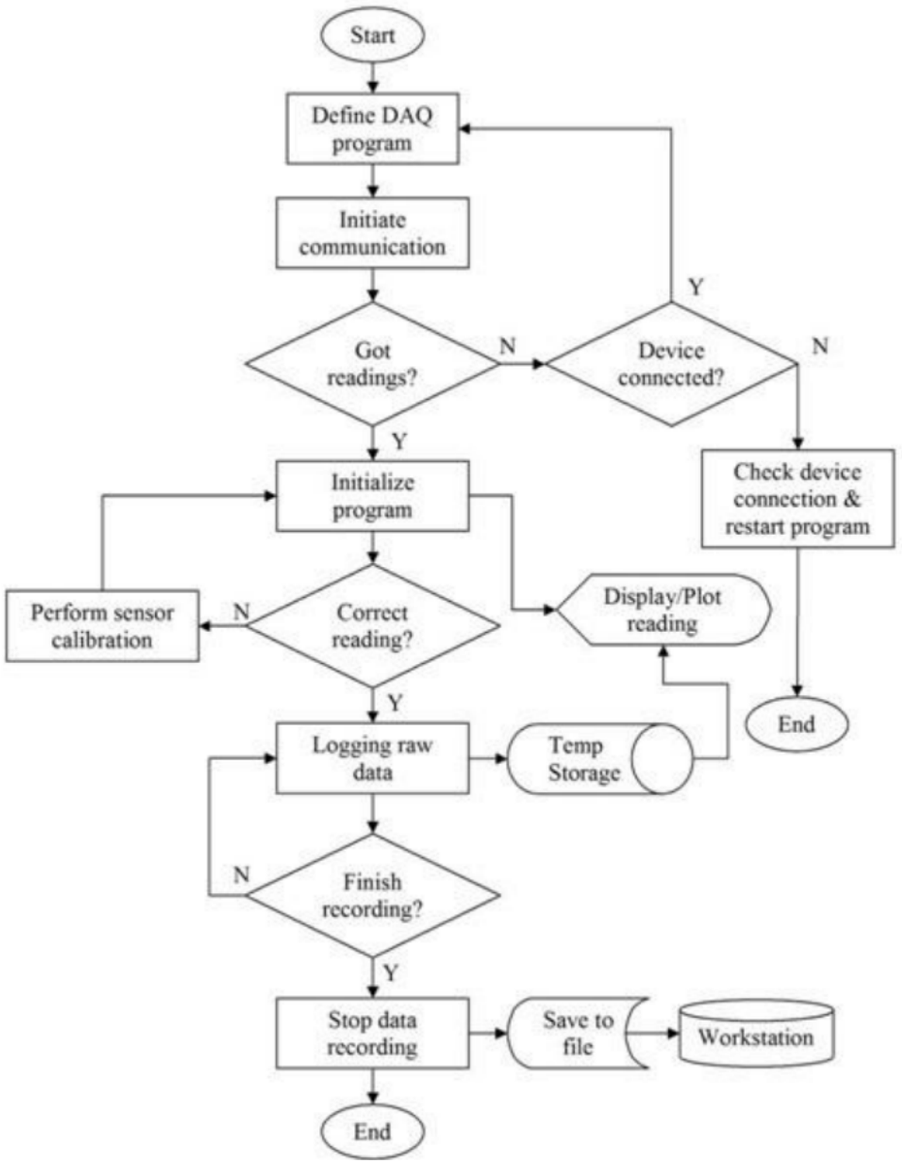
3 Procedural abstraction

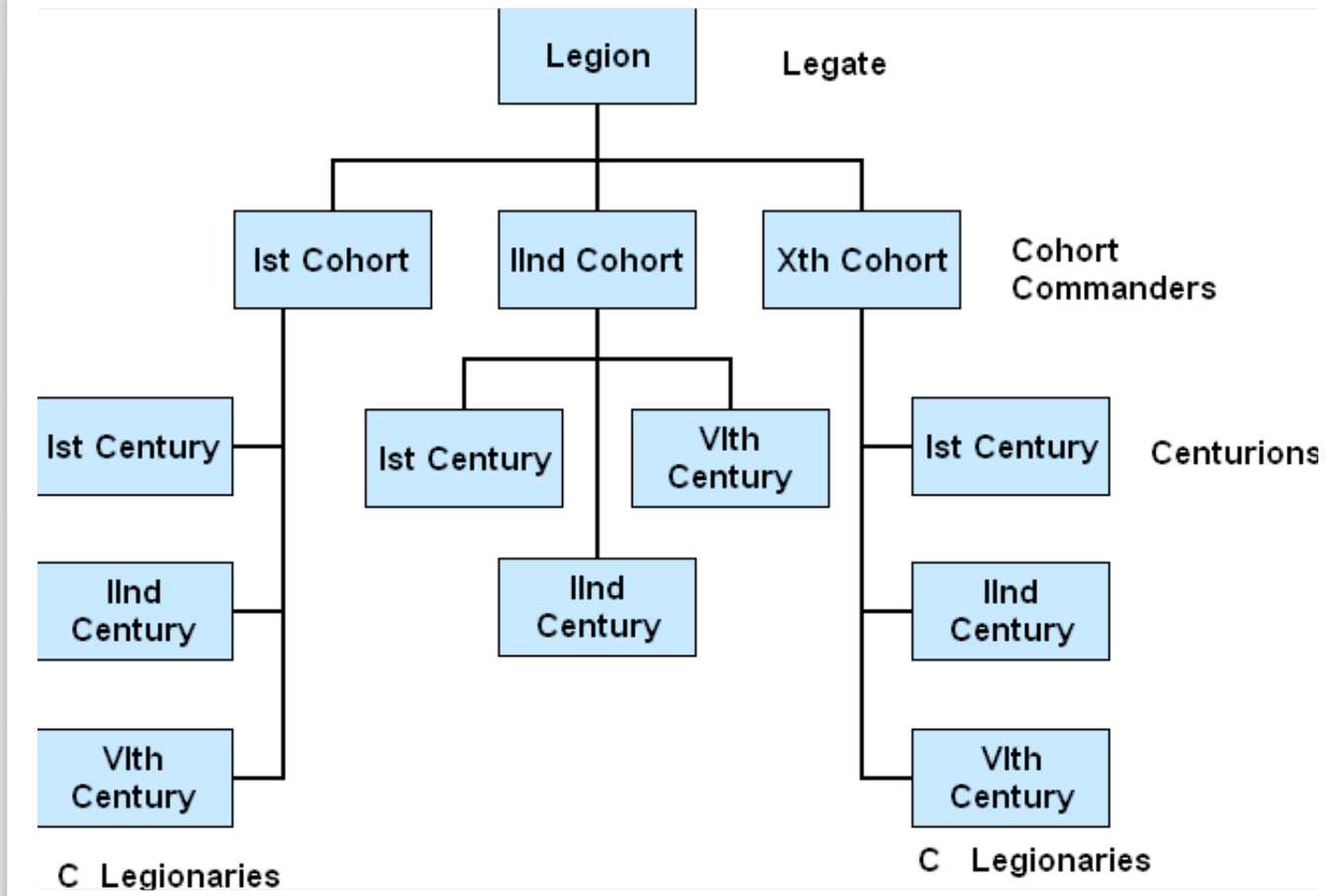
1. Top-down design
2. Procedures and functions. Parameters. Example. Procedure and function declarations. Formal parameters. Subprogram calling. Actual parameters. By value and by reference parameter. Interface. Modularity criteria. Global and local variables. Side effects. Preconditions and postconditions. Exceptional conditions treatment
3. Recursion. Recursion concept. Examples. Recursion versus iteration

1. Top-down design

- How to tackle **complexity**?



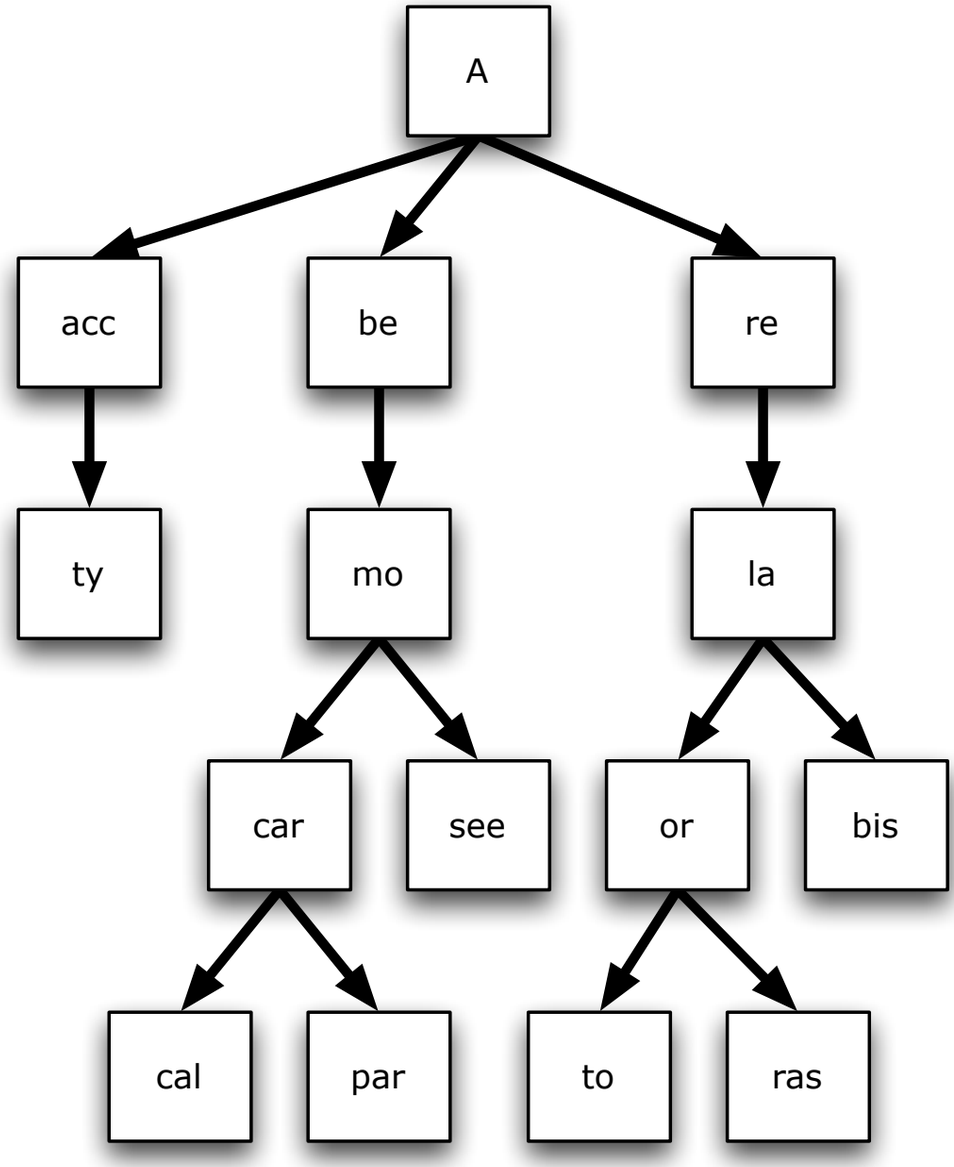




1. Top-down design

- Decomposition into subproblems

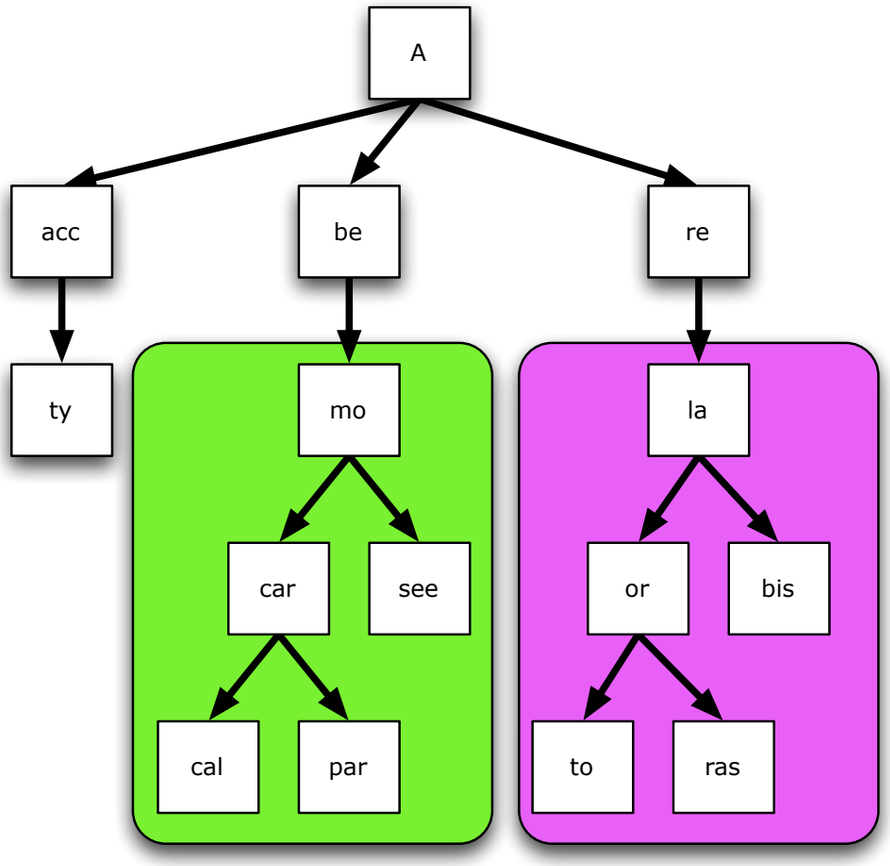
divide and conquer



1. Top-down design characteristics

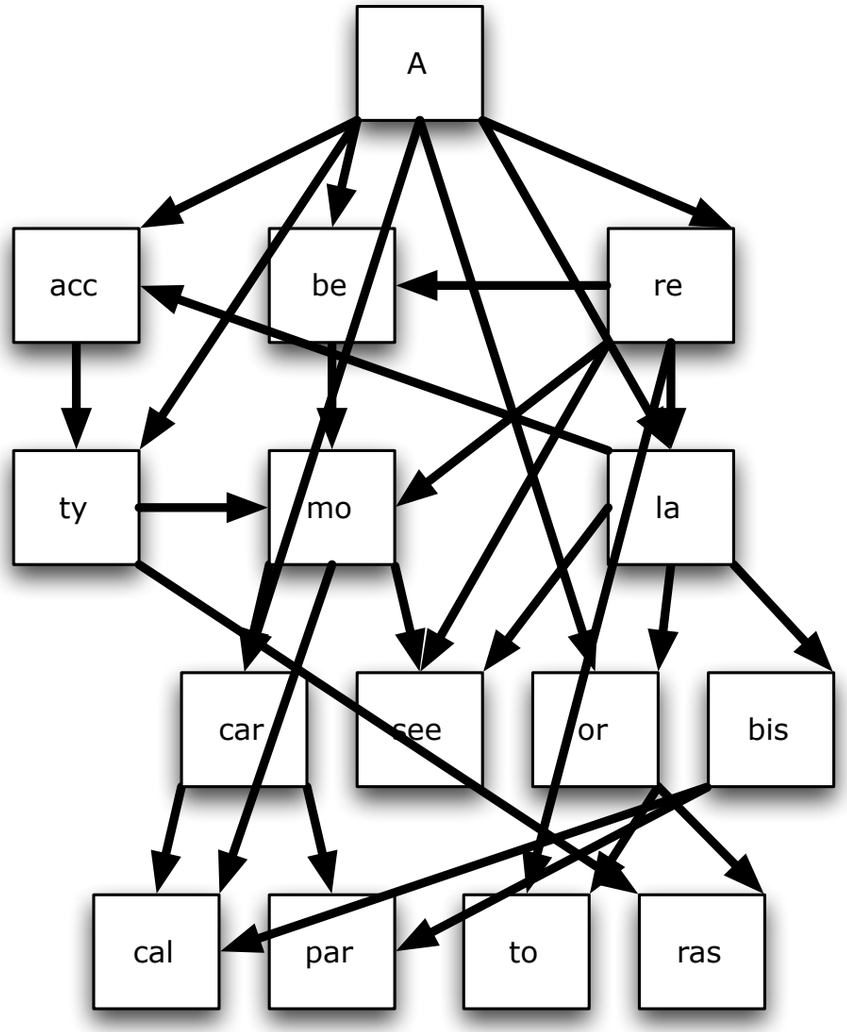
👍 Independency

allows hierarchy



versus

👎 Subproblems **coupling** *is bad*



- Hierarchical organisation allows easier subdivision

1. Top-down design, what is it

- Each subpart is a **subproblem** to solve
  - Subproblems are easier to solve
  - Decomposing into subproblems makes the main problem easier to read
- …easier to modify
- …easier to reuse
- …etc…

2. Declaration

How to build a new subprogram

type | name | (formal parameters…) | {code}

```
int fact(int n)
{
    int r = 1;
    for (int i = 2; i < n; ++i)
        r *= i;
    return r;
}
```

1. Basic example (wo prototypes)

```
#include <iostream>
using namespace std;

int fact(int n)
{
    int f = 1;
    for ( int i = 2; i <= n; ++i )
        f *= i;
    return f;
}

int main()
{
    cout << "n?: ";
    int n;
    cin >> n;
    for ( int i = 0; i < n; ++i )
        cout << i << ": " << fact(i) << endl;

    return 0;
}
```

2. Nesting NOT allowed

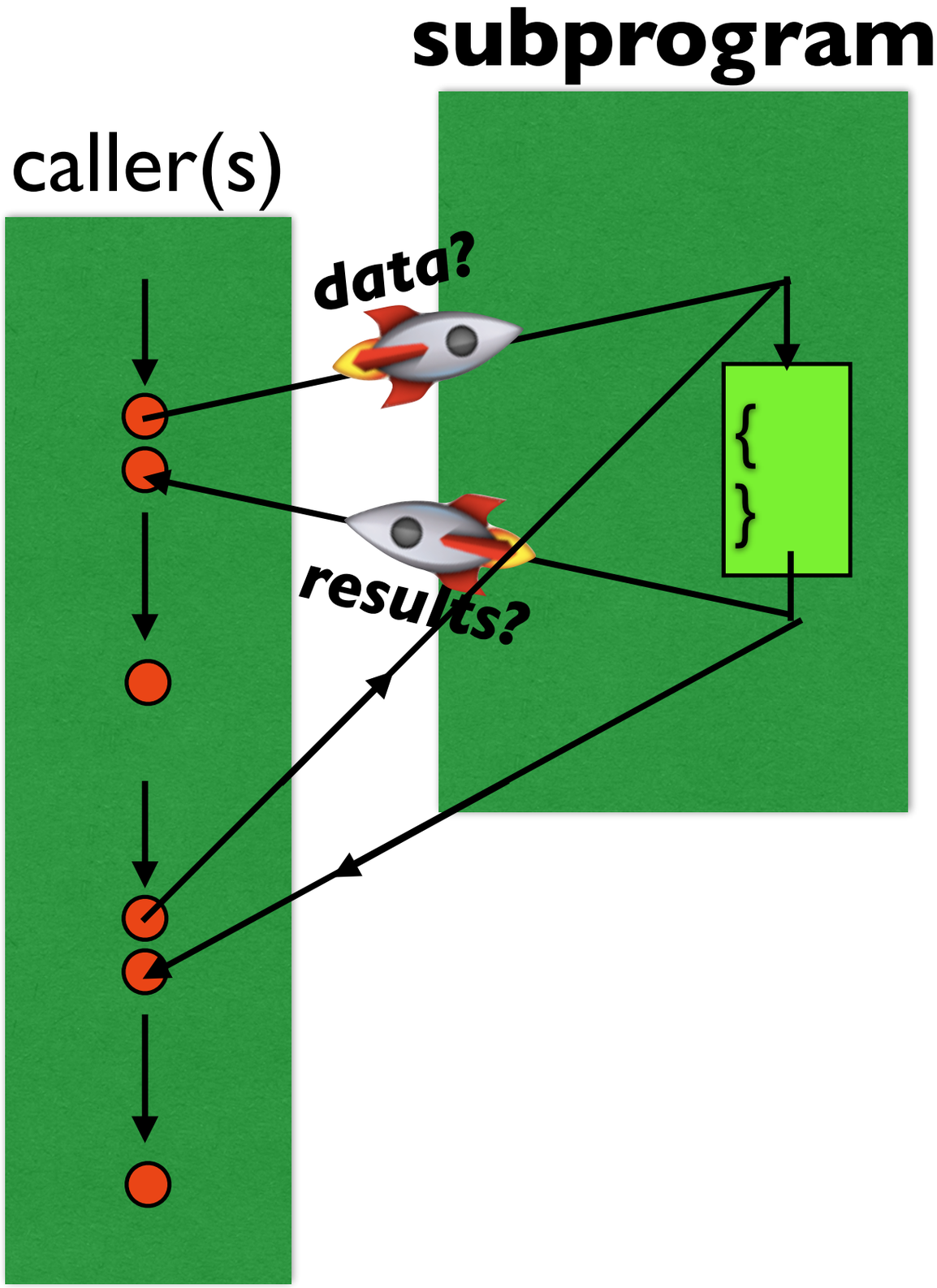
- **This must be written <u>BEFORE</u>** `int main()`**...**
- **never inside another function,**
- ***<u>NO NESTING ALLOWED</u>***

```
#include <iostream>
using namespace std;

int main()
{
    int fact(int n)
    {
        int f = 1;
        for ( int i = 2; i <= n; ++i )
            f *= i;
        return f;
    }
    cout << "n?: ";
    int n;
    cin >> n;
    for ( int i = 0; i < n; ++i )
        cout << i << ": " << fact(i) << endl;

    return
}
```

2. Execution

1. The caller prepares and sends **data** to the subprogram (*return address included*!)
2. Jump to the subprogram code
3. When subprogram finishes, control is returned to the next instruction after the call

This is repeated in each call



1. Parameters

- When call, caller send values as **parameter**. Parameters are **copied** onto the formal ones…

```
int combinatorial(int n, int p) {…}

void printComplex(float real, float im) {…}

void printlns(int n=1) {…}

float exseries(float x, float precis) {…}
```

```
int main()
{
    …
    cout << combinatorial(a, 2);
    printComplex(2+3.0, p_im);
    printComplex(x, y);
    printlns();
    printlns(3);
    r = exseries(0.5, 0.00001);
    …
```

2. Places of declaration

declare before use

```
sub01

sub02

…

sub99

int main()
```

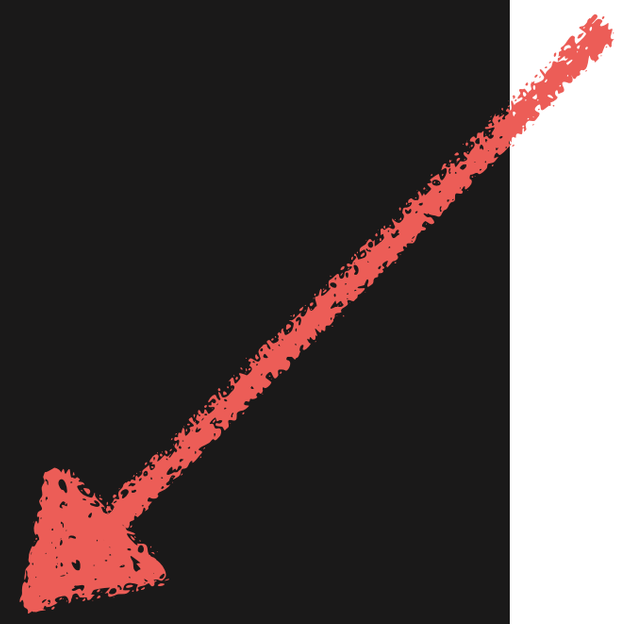
Subprograms (*as everything else*) **must always** be declared **before** their use

…this implies that all of them in a program should be declared before **main()** and… ordered following possible dependences

1. Top-down design

```
#include <iostream>
using namespace std;

int fact(int n)
{
    int f = 1;
    for ( int i = 2; i <= n; ++i )
        f *= i;
    return f;
}

int main()
{
    cout << "n?: ";
    int n;
    cin >> n;
    for ( int i = 0; i < n; ++i )
        cout << i << ": " << fact(i) << endl;

    return 0;
}
```

before using it



2. Prototypes

The strict order and positioning of subprograms can be ***relaxed*** through **prototype** declaration

interface: is the name given to them in general

2. Prototypes

prototypes

```
// subs.cpp
// juanfc 2017-11-13
// Many subprograms

#include <iostream>
using namespace std;

void  sub01(string msg);
int   sub02(int n);
float sub03(float x, int t);
float sub04(float x);

int main()
{
    sub01("Hello");
    cout << sub02(33) << endl;
    cout << sub03(2.1, 3) +
            sub04(3.14) << endl;
    return 0;
```

```
}


void  sub01(string msg)
{
    // ...
}
int   sub02(int n)
{
    // ...
    return res;
}
float sub03(float x, int t)
{
    // ...
    return res;
}
float sub04(float x, float d)
{
    // ...
    return res;
}
```

2. Prototypes

A **prototype** is

a subprogram

declaration but

without body

```
#include <iostream>
using namespace std;

int fib(int n); // prototype

int main()
{
    int n;
    cout << "Please enter n: ";
    cin >> n;
    cout << "Fib: " << fib(n) << endl;
    return 0;
}

int fib(int n)
{
    int a=0, b=1, f=a+b;
    for ( int i = 3; i <= n; ++i ) {
        a = b;
        b = f;
        f = a + b;
    }
    return f;
}
```

EXAMPLE

$$\binom{m}{n} = \frac{m!}{n!\,(m-n)!}$$

You have to compute the same thing three times, and multiply and divide the results

Here a function is of help:

```
fact(x) / fact(x-y) / fact(y);
```

Example of correct calls

$$\binom{m}{n} = \frac{m!}{n!\,(m-n)!}$$

observe the order
of the writing

```
#include <iostream>
using namespace std;

// prototypes
int comb(int x, int y);

int main()
{
    int x, y;
    cout << "Enter x y: " << endl;
    cin >> x >> y;
    cout  << x << " choose " << y << ": "
          << comb(x, y) << endl;
    return 0;
}

int fact(int n);
int comb(int x, int y)
{
    return fact(x)/fact(x-y)/fact(y);
}

int fact(int n)
{
    int f=1;
    for ( int i = 2; i <= n; ++i )
        f *= i;
    return f;
}
```

There are 2 types of *subprograms*:

- **Procedures**
- **Functions**

1. Procedures and functions

procedure

```
void name(param, param,…)
{
  // code
}
```

function

```
type name(param, param,…)
{
  // code
  return something_of_type
}
```

1. Parameters

1. Parameters are the way subprograms have to **receive** values from the caller
2. The variables **declared** inside the parenthesis are called → **formal parameters**
3. Real or **actual parameters** (in the caller) are **copied onto** formal parameters
4. These variable are **local** to the subprogram

```
int combinatorial(int n, int p) {…}

void printComplex(float real, float im) {…}

void printlns(int n=1) {…}

float exSeries(float x, float precis) {…}
```

```cpp
#include <iostream>
using namespace std;

int fact(int n);

int main()
{
    cout << "n?: ";
    int n;
    cin >> n;
    for ( int i = 0; i < n; ++i )
        cout << i << ": " << fact(i) << endl;

    return 0;
}

int fact(int n)
{
    int f = 1;
    for ( int i = 2; i <= n; ++i )
        f *= i;
    return f;
}
```

formal

each time main calls *fact*…

the new value of *i* is **copied** into *n*

actual or real

Short list of popular standard functions

Name	Descrip.	Args	Returns	Example	result	header
sqrt	√x	double	double	**sqrt**(4.5)	2.12132	cmath
pow	x^y	double, double	double	**pow**(2.2, 3.1)	11.5215	cmath
abs	|x|	int	int	**abs**(–7)	7	cstdlib
labs	|x|	long	long	**labs**(–7)	7	cstdlib
fabs	|x|	double	double	**fabs**(–7.5)	7.5	cmath
ceil	⌈x⌉	double	double	**ceil**(2.01)	3.0	cmath
floor	⌊x⌋	double	double	**floor**(3.99)	3.0	cmath
srand	seed rand	-	-	**srand**()	-	cstdlib
rand	rand gen	-	int	**rand**()	0–RAND_MAX*	cstdlib

*) any integer between 0 and RAND_MAX

Exercises

Build the ***interfaces*** for the next subprograms:

(1) Compute the surface of a circle

(2) … the greatest of three numbers

(3) that returns the ASCII code of a letter

(4) given a number, returns the letter in that position in the alphabet

(5) given two dates, return the distance in days between them (we want only the interface)

(6) guess whether a number is even or odd

(7) guess whether a date is or not a leap year

Each subprogram has its own variables

- **Curly-braces {} enclose *scopes***
- **Things inside are hidden from other scopes**

- They isolated each {piece of code} from the {others}
- So… how to send back to the caller more pieces of information?

By reference parameters &

2. Parameter contents

- What do you get from the next program?

```
void swap(int a, int b);

int main()
{
    int x, y;
    cout << "Enter two numbers:
    cin >> x >> y;
    swap(x, y);
    cout << x << ", " << y << endl;
    return 0;
}

void swap(int a, int b)
{
    int t = a;
    a = b;
    b = t;
}
```

As parameter are copied, changes are done inside, in local variables, and do not affect actual vars in main

The contents are not swapped

2. Parameter by &reference

- Many times we need
  more than one result from a call

- **return** is capable of returning only 1 thing

- How to return 2, 3, or more things from a function?

*the only solution is through the **parameters***

2. Parameter by reference

&

- C is unable to **return** more than 1 thing in each call, so

- C uses a different mechanism for dealing with more than one output

```
void swap(int& a, int& b);

int main()
{
    int x, y;
    cout << "Enter two numbers: ";
    cin >> x >> y;
    swap(x, y);
    cout << x << ", " << y << endl;
    return 0;
}

void swap(int& a, int& b)
{
    int t = a;
    a = b;
    b = t;
}
```

2. You need to *return* more than 1 value

- a second degree equation is characterised by three numbers a, b, and c:

$$ax^2 + bx + c = 0$$

build the interface for a procedure to solve it returning its two possible answers

What if we wanted to get any of the 6 possible outputs from this call…?

2. Parameters by ref

- Parameters by ref need real parameters to be exactly the same type as formal parameters

- This is quite restrictive

```
void swap(int& a, int& b);

int main()
{
    int x, y;
    cout << "Enter two numbers: ";
    cin >> x >> y;
    swap(x, y);
```

2. Parameters by ref + and – aspects

In general:

- Vulnerable to lateral effect issues
- They need actual params to be vars of **identical** type
- They do not admit constants or expressions as actual params → *are more limited in their use*

- Faster especially for large data (no copying needed!)
- Allow returning 2 or more items

by reference parameters, when?

Only use then when you need to return **more than one** value

- *That is, do not use by-ref & parameters, except when really needed*