

Functions and Procedures

Code Capsules in C++

Professors de Pro1

Dept. of Computer Science, UPC

Outline

- 1 What Are Code Capsules?
- 2 Functions vs. Procedures
- 3 Examples
- 4 Parameter Passing
- 5 Reference Parameter Examples
- 6 Summary

Programming Resources: Code Capsules

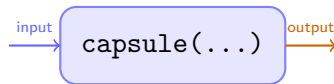
Idea

A **code capsule** encapsulates a reusable piece of logic.

Once defined, you can **call** it whenever needed — without rewriting it.

Two kinds of capsules:

- **Functions** compute a value
- **Procedures** perform a task



Why?

Avoid repetition. Improve readability.
Divide and conquer.

Definition

A function **computes and returns a single value**.

- The **return** instruction is **mandatory**
- Can be used inside any **expression**
- The return type replaces **void**

Typical examples

- Square root of a number
- Maximum of two integers
- Sum of digits of a number
- Is n a power of 2?

Rule in Pro1

Functions return **exactly one value**.

Procedures

Definition

A procedure **performs a task** that cannot be expressed as a single value.

- **No** return instruction
- Return type is **void**
- Must be called as a **standalone statement**

Typical examples

- Swap two variable values
- Draw a figure on screen
- Sort three numbers
- Compute quotient *and* remainder

Rule in Pro1

Use a procedure when you need to produce **two or more results**.

Quick Comparison

Feature	Function	Procedure
Return type	int, bool, ...	void
return keyword	Mandatory	Not used
Number of results	Exactly one	Zero, one, or more
How it is called	Inside an expression	Standalone statement
Parameter passing	Mainly by value	By value and by reference

Example 1 — Function: maximo

```
1 // pre:  ---
2 // post: returns the max of a and b
3 int maximo(int a, int b) {
4     if (a >= b) return a;
5     else      return b;
6 }
```

Anatomy

- **Head:** name + formal parameters
- **Body:** logic + return
- Return type: int

Calling it

```
cout << maximo(n+5, m-3);
```

Used directly inside an expression.

Example 2 — Procedure: felicitar

```
1 // pre:  ---
2 // post: congratulates nombre
3 void felicitar(string nombre) {
4     cout << "Feliz cumpleannos "
5           << nombre
6           << ", que pases un buen dia!"
7           << endl;
8 }
```

Anatomy

- Return type: void
- No return statement
- Does something visible (output)

Calling it

```
felicitar("Joan");
```

Standalone statement — not inside an expression.

Example 3 — Function: es_pot_dos

```
1 // pre:  a >= 1
2 // post: true iff a is a power of 2
3 bool es_pot_dos(int a) {
4     while (a%2 == 0) a = a/2;
5     return a == 1;
6     // avoid:
7     // if (a==1) return true;
8     // else return false;    <- redundant
9 }
```

Key points

- Return type: bool
- return a == 1 is concise and correct
- Precondition: a >= 1

Calling it

```
if (es_pot_dos(n))...
```

Input, Output, and Input/Output Parameters

Input

Data **flows in**.

The function reads the value but does not change the caller's variable.

Output

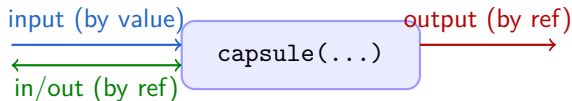
Data **flows out**.

The function writes a result into the caller's variable.

Input/Output

Data **flows both ways**.

The function reads *and* modifies the caller's variable.



Call-by-Value (Input Parameters)

How it works

Sets up a **one-way** channel from the call point to the formal parameter.

The expression at the call site is evaluated; its value is *copied* into the parameter.

Effect

Changes to the formal parameter **do not affect** the caller's variable.

Examples

```
1 // n=10, m=20
2 cout << maximo(n+5, m-3);
3 // --> prints 17
4
5 felicitar("Joan");
6
7 if (es_pot_dos(n)) ...
```

Call-by-Reference (Output / In-Out Parameters)

How it works

Sets up a **two-way** channel. The formal parameter becomes an *alias* for the variable at the call site.

Changes inside the body **are visible** to the caller.

Pro1 rule

Output and In-Out parameters are used **only in procedures**.

Syntax: add &

```
1 void swap(int& a, int& b);  
2 //           ^           ^  
3 //           by-reference params
```

At the call site

Pass **variables** (not expressions):

```
swap(x, y);
```

Procedure: swap

```
1 // pre:  ---
2 // post: a and b have swapped values
3 void swap(int& a, int& b) {
4     int temp = a;
5     a = b;
6     b = temp;
7 }
```

Why a procedure?

Two variables are modified. Cannot be expressed as a single return value.

Parameters

Both a and b are **input/output**: they are read and then overwritten.

Procedure: division_entera

```
1 // pre:  a >= 0 and b > 0
2 // post: c and r are the quotient and remainder of a divided by b
3 void division_entera(int a, int b, int& c, int& r) {
4     c = a/b;
5     r = a%b;
6 }
```

Parameter roles

- a, b: **input** (by value)
- c, r: **output** (by reference)

Valid calls

```
1 division_entera(9834, 46, quotient,
2                 remainder);
2 division_entera(n, m+1, c, r);
```

Summary

Functions

- Compute **one** value
- Non-void return type
- return mandatory
- Called inside expressions
- Parameters: call-by-value

Procedures

- Perform a task / multiple results
- Return type void
- No return
- Called as standalone statements
- Can use call-by-reference

Golden rule

Always write **preconditions** and **postconditions** as comments before any capsule.