



SAPIENZA
UNIVERSITÀ DI ROMA

Algorithms

Volume 2 — Programming languages

From pseudo-code to programs: three programming languages, C++ (compiled), Python (interpreted) and Java (compiled to bytecode and run by a virtual machine). For each: variables and data types, input and output, operators, conditional statements, loops, arrays and lists, functions, bitwise operators and a complete algorithm, with the comparison of the three languages.

Fabio Furini

fabiofurini.github.io

Department of Computer, Control and Management Engineering — Sapienza University of Rome

Online version and downloadable material:

fabiofurini.github.io/algorithms/
github.com/fabiofurini/algorithms

Academic year 2026–2027

Contents

6	Introduction to C++	3
	Introduction	3
6.1	Main characteristics of the programming language	4
6.2	Variables and data types	5
6.3	Input-output stream	6
6.4	Operators	7
6.5	Conditional statements	12
6.6	Loops	14
6.7	Arrays	16
6.8	The <code>main</code> and custom functions	19
6.9	A complete algorithm: the maximum value in an array	21
6.10	Bitwise operators	23
7	Introduction to Python	31
	Introduction	31
7.1	Main characteristics of the programming language	32
7.2	Variables and data types	34
7.3	Input and output	35
7.4	Operators	36
7.5	Conditional statements	42
7.6	Loops	44
7.7	Lists	46
7.8	Functions	49
7.9	A complete algorithm: the maximum value in an array	51
7.10	Bitwise operators	54
7.11	Python and C++ side by side	55
8	Introduction to Java	56
	Introduction	56
8.1	Main characteristics of the programming language	57
8.2	Variables and data types	58
8.3	Input and output	60
8.4	Operators	61



8.5	Conditional statements	66
8.6	Loops	68
8.7	Arrays	69
8.8	Methods (functions)	72
8.9	A complete algorithm: the maximum value in an array	75
8.10	Bitwise operators	78
8.11	Java, C++ and Python side by side	79

Introduction to C++

Introduction

In the chapter «Pseudo-code» algorithms are written in a language meant for humans: it is precise, but it is not executed by a computer. To run an algorithm we have to translate it into a **programming language**. In this chapter we introduce the programming language C++, which we use to implement the algorithms of the course.

The chapter follows the same path as the chapter «Pseudo-code»:

- the main characteristics of the language and the structure of a program;
- variables, data types and the input-output stream;
- operators, conditional statements and loops;
- arrays, whose size can be fixed or decided during the execution;
- functions, with the `main` function as the entry point of every program;
- a complete algorithm, the search of the maximum value in an array, written in pseudo-code and in C++;
- the bitwise operators, which act directly on the binary representation of integer numbers.

Every fragment of code of the chapter has been compiled with the `g++` compiler (standard C++17) and executed: the outputs reported in the examples are the ones printed by the programs.

In pseudo-code the entries of an array $a[1..n]$ are indexed from 1 to n ; in C++ the entries of an array of size n are indexed from 0 to $n - 1$. The entry $a[i]$ of the pseudo-code corresponds to the entry $a[i-1]$ of the C++ code.

6.1 Main characteristics of the programming language

The main characteristics of C++ are the following.

- C++ is an **imperative** language that gives direct access to the memory and to the hardware resources of the computer. It is highly efficient for low-level system programming while also supporting high-level abstractions, making it versatile for both performance-critical applications and modern software development.
- C++ supports **functions**, which enable a structured and modular approach to programming. Functions allow code reuse and better organization.
- The C++ programming language includes a predefined set of **reserved symbols** used to declare variables, define functions and specify executable instructions. The **source code** consists of a sequence of **atomic operations** executed in order, one after the other, operating on variables of specific **data types**.
- The source code in C++ can contain **comments**, which serve as documentation for the code and are ignored by the compiler. In C++, a single-line comment starts with the reserved symbol “//”. Multi-line comments are enclosed between the reserved symbols “/*” and “*/”.

Example 6.1: comments in C++

This is an example of a single-line comment:

```
1 // This is a comment line
```

This is an example of a multi-line comment:

```
1 /* This is a
2    multi-line comment */
```

6.1.1 The structure of a program

Before describing the single instructions, we look at the skeleton that every program shares. A C++ program is a text file (with extension `.cpp`) that is translated by a **compiler** into an executable file. The execution of the program starts from the **main** function, the entry point of every program.

Example 6.2: the skeleton of a program

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     // The instructions of the program are written here
8     cout << "Hello, world!" << endl;
```

```
9  
10     return 0;  
11 }
```

If the source code is saved in the file `hello.cpp`, it is compiled and executed with the commands

```
g++ -Wall -std=c++17 -o hello hello.cpp  
./hello
```

and the output is

```
Hello, world!
```

In the following examples we write only the instructions that matter: unless they contain a complete program, the fragments of code are meant to be placed inside the curly braces of `main`.

Each instruction of a C++ source code must end with the reserved symbol “;”.

6.2 Variables and data types

The source code operates on **variables**, and every variable has a **data type**.

- Information is categorized into different **data types**. The four fundamental families of data types are:
 1. The data type “`int`” represents an integer, which is a whole number written without a decimal point. This includes positive and negative whole numbers, as well as zero.
 2. The data type “`bool`” represents a Boolean value: `true` or `false`.
 3. The data type “`float`” represents a floating-point number, which includes numbers written with a decimal point. The data type “`double`” represents a floating-point number with double precision, providing greater accuracy and range than `float`.
 4. The data type “`char`” represents a single character, such as a letter, digit, or symbol, enclosed in single quotes (e.g., `'A'`, `'7'`, `'!'`).
- A sequence of characters enclosed in double quotes (e.g., `"Ada"`) is stored in a variable of type “`string`”, provided by the standard library `<string>`.
- A variable acts as a container that stores a value of a specific data type for later use. Each variable has a unique name and holds a single value at any given time.
- Variables are assigned values using the assignment operator “`=`”. When a new value is assigned, the previous value is overwritten.

Variables must be declared before use, specifying their data type.

Example 6.3: variable declarations and value assignments

```

1 int age = 25; // Integer variable
2 bool isRaining = false; // Boolean variable
3 float smallNumber = 3.14; // Floating-point variable
4 double largeNumber = 123456.789; // Double precision floating-point variable
5 char character = 'a'; // Char variable
6 string name = "Ada"; // String variable
7
8 age = 26; // The previous value 25 is overwritten
9 cout << age << " " << isRaining << " " << smallNumber << " "
10    << largeNumber << " " << character << " " << name << endl;

```

Output:

26 0 3.14 123457 a Ada

The Boolean value `false` is displayed as 0, and the number 123456.789 is displayed as 123457: by default, `cout` displays at most six significant digits (the value stored in the variable does not change, and the number of displayed digits can be chosen, as we see now).

6.3 Input-output stream

To display the values of the variables on the screen, a program uses the **input-output stream** library.

- The include “`#include <iostream>`” allows to use the input-output stream library in C++.

Example 6.4: include

```

1 #include <iostream>

```

- The values of variables can be displayed as output text on the screen using the output stream “`std::cout`”. The insertion operator “`<<`” is used to send data to the output stream.
- Multiple values can be displayed in a single `cout` statement by chaining them with the “`<<`” operator. Text should be enclosed in double quotes “`”`”, while variable values are written directly.
- The manipulator “`std::endl`” is used with `std::cout` to insert a new line on the screen and flush the output buffer.

Example 6.5: screen printings

```

1 int n = 7;
2 std::cout << "The value of the variable is " << n << std::endl;

```

Output:

The value of the variable is 7

- The statement “`using namespace std;`” simplifies the use of standard library functions by eliminating the need to prefix them with `std::`. From now on we always use it, and we write `cout` and `endl`.
- The include “`#include <iomanip>`” gives the manipulators “`fixed`” and “`setprecision(k)`”. When they are written together in a `cout` statement, they display the floating-point numbers that follow with exactly k digits after the decimal point.

6.4 Operators

Operators are symbols that perform operations on variables and values. They are classified, according to their functionality, into five main categories: arithmetic, assignment, increment and decrement, relational and logical operators.

6.4.1 Arithmetic operators

Arithmetic operators, used to perform basic mathematical operations:

- `+` (Addition) $\rightsquigarrow$ Adds two values.
- `-` (Subtraction) $\rightsquigarrow$ Subtracts one value from another.
- `*` (Multiplication) $\rightsquigarrow$ Multiplies two values.
- `/` (Division) $\rightsquigarrow$ Divides one value by another.
- `%` (Modulus) $\rightsquigarrow$ Returns the remainder of a division.

Example 6.6: arithmetic operators

```
1 int a = 10, b = 3;
2
3 cout << "Addition: " << (a + b) << endl;
4 cout << "Subtraction: " << (a - b) << endl;
5 cout << "Multiplication: " << (a * b) << endl;
6 cout << "Division: " << (a / b) << endl; // Integer division
7 cout << "Modulus: " << (a % b) << endl; // Remainder of division
```

Output:

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3
Modulus: 1
```

- The division operator `/` performs integer division when used with integers: the result $10 / 3$ is 3. To obtain a floating-point result, at least one operand must be of data type `float` or `double`. For example, $10.0 / 3$ returns 3.33333 instead of 3.

- A value of type `int` is converted into a value of type `double` by writing “(double)” before it (**type conversion**). For example, if `a` and `b` are variables of type `int`, the expression `(double)a / b` performs the floating-point division.

Example 6.7: integer and floating-point division

```

1 int a = 10, b = 3;
2
3 cout << a / b << endl;           // Integer division: 3
4 cout << 10.0 / 3 << endl;        // Floating-point division
5 cout << (double)a / b << endl;   // Conversion of a to double
6 cout << fixed << setprecision(2) << (double)a / b << endl; // Two decimals

```

Output (the last line requires `#include <iomanip>`):

```

3
3.33333
3.33333
3.33

```

6.4.2 Assignment operators

Assignment operators, used to assign values to variables:

- `=` (Assignment) $\rightsquigarrow$ Assigns a value to a variable.
- `+=` (Addition assignment) $\rightsquigarrow$ Adds a value to a variable and assigns the result.
- `-=` (Subtraction assignment) $\rightsquigarrow$ Subtracts a value from a variable and assigns the result.
- `*=` (Multiplication assignment) $\rightsquigarrow$ Multiplies a variable by a value and assigns the result.
- `/=` (Division assignment) $\rightsquigarrow$ Divides a variable by a value and assigns the result.
- `%=` (Modulus assignment) $\rightsquigarrow$ Applies the modulus operation and assigns the result.

Example 6.8: assignment operators

```

1 int x = 10; // Assignment
2 cout << "Initial value: " << x << endl;
3
4 x += 5; // Equivalent to: x = x + 5
5 cout << "After += 5: " << x << endl;
6
7 x -= 3; // Equivalent to: x = x - 3
8 cout << "After -= 3: " << x << endl;
9
10 x *= 2; // Equivalent to: x = x * 2
11 cout << "After *= 2: " << x << endl;

```

```

12
13 x /= 4; // Equivalent to: x = x / 4
14 cout << "After /= 4: " << x << endl;
15
16 x %= 3; // Equivalent to: x = x % 3
17 cout << "After %= 3: " << x << endl;

```

Output:

```

Initial value: 10
After += 5: 15
After -= 3: 12
After *= 2: 24
After /= 4: 6
After %= 3: 0

```

6.4.3 Increment and decrement operators**Increment and decrement operators:**

- ++ (Increment) $\rightsquigarrow$ Increases the value of a variable by 1.
- -- (Decrement) $\rightsquigarrow$ Decreases the value of a variable by 1.
- The increment operator ++ can be used in two forms:
 - pre-increment (“++x”): increments the value before returning it;
 - post-increment (“x++”): returns the value first, then increments it.

The same logic applies to the decrement operator (--x and x--).

Example 6.9: increment and decrement operators

```

1 int x = 5;
2
3 cout << "Initial value: " << x << endl; // Displays 5
4
5 cout << "Pre-increment (++x): " << ++x << endl; // Displays 6
6
7 cout << "Post-increment (x++): " << x++ << endl; // Displays 6
8 cout << "Value after post-increment: " << x << endl; // Displays 7
9
10 cout << "Pre-decrement (--x): " << --x << endl; // Displays 6
11
12 cout << "Post-decrement (x--): " << x-- << endl; // Displays 6
13 cout << "Value after post-decrement: " << x << endl; // Displays 5

```

Output:

```

Initial value: 5
Pre-increment (++x): 6
Post-increment (x++): 6
Value after post-increment: 7

```

```
Pre-decrement (-x): 6
Post-decrement (x-): 6
Value after post-decrement: 5
```

6.4.4 Relational operators

Relational operators, used to compare values and return a Boolean result:

- `==` (Equal to) $\rightsquigarrow$ Checks if two values are equal.
 - `!=` (Not equal to) $\rightsquigarrow$ Checks if two values are different.
 - `<` (Less than) $\rightsquigarrow$ Checks if one value is smaller than another.
 - `>` (Greater than) $\rightsquigarrow$ Checks if one value is larger than another.
 - `<=` (Less than or equal to) $\rightsquigarrow$ Checks if one value is smaller than or equal to another.
 - `>=` (Greater than or equal to) $\rightsquigarrow$ Checks if one value is larger than or equal to another.
- Relational operators return a Boolean result:
 - `true` (displayed as 1) if the condition is satisfied;
 - `false` (displayed as 0) if the condition is not satisfied.

These operators are commonly used in **conditional statements** and **loops**.

Example 6.10: relational operators

```
1 int a = 10, b = 5;
2
3 cout << "a == b: " << (a == b) << endl; // Equal to
4 cout << "a != b: " << (a != b) << endl; // Not equal to
5 cout << "a < b: " << (a < b) << endl;   // Less than
6 cout << "a > b: " << (a > b) << endl;   // Greater than
7 cout << "a <= b: " << (a <= b) << endl; // Less than or equal to
8 cout << "a >= b: " << (a >= b) << endl; // Greater than or equal to
```

Output:

```
a == b: 0
a != b: 1
a < b: 0
a > b: 1
a <= b: 0
a >= b: 1
```

6.4.5 Logical operators

Logical operators, used for logical operations, often in conditional statements:

- `&&` (Logical AND) $\rightsquigarrow$ Returns **true** if both conditions are true.
 - `||` (Logical OR) $\rightsquigarrow$ Returns **true** if at least one condition is true.
 - `!` (Logical NOT) $\rightsquigarrow$ Reverses the Boolean value.
- Like the relational operators, logical operators return Boolean values, **true** (1) or **false** (0), and are commonly used in **conditional statements** and **loops**.

Example 6.11: logical operators

```

1 bool condition1 = true;
2 bool condition2 = false;
3
4 cout << "condition1: " << condition1 << endl;
5 cout << "condition2: " << condition2 << endl;
6
7 // Logical AND: true only if both conditions are true
8 cout << "condition1 && condition2: " << (condition1 && condition2) << endl;
9
10 // Logical OR: true if at least one condition is true
11 cout << "condition1 || condition2: " << (condition1 || condition2) << endl;
12
13 // Logical NOT: reverses the Boolean value
14 cout << "!condition1: " << !condition1 << endl;
15 cout << "!condition2: " << !condition2 << endl;
```

Output:

```

condition1: 1
condition2: 0
condition1 && condition2: 0
condition1 || condition2: 1
!condition1: 0
!condition2: 1
```

6.5 Conditional statements

The relational and logical operators produce the conditions that decide which instructions are executed.

- Conditional statements allow to execute different blocks of code based on specific conditions. The `if` statement is the most fundamental control structure of this type.
- The `if` statement evaluates a condition inside parentheses (). If the condition is **true**, the block of code inside the curly braces {} is executed. If the condition is **false**, the program skips the block and executes the block of code inside the curly braces {} of the `else` part, if present.

Example 6.12: conditional statement

```

1 int number = 10;
2
3 // Check if number is positive
4 if (number > 0)
5 {
6     cout << "The number is positive." << endl;
7 }
8 else
9 {
10    cout << "The number is non-positive." << endl;
11 }
```

Output:

The number is positive.

6.5.1 Ternary operator (conditional operator)

- The **ternary operator** `?:` provides a compact way to write simple `if-else` statements.
- It is the only operator in C++ that takes three operands.
- Syntax: `condition ? value_if_true : value_if_false`
- If the condition is **true**, the expression evaluates to `value_if_true`; otherwise, it evaluates to `value_if_false`.

Example 6.13: ternary operator

```

1 int number = 10;
2
3 // Using if-else statement
4 string result1;
5 if (number > 0)
6     result1 = "positive";
7 else
```

```

8     result1 = "non-positive";
9
10    cout << "The number is " << result1 << endl;
11
12    // Using ternary operator (compact form)
13    string result2 = (number > 0) ? "positive" : "non-positive";
14
15    cout << "The number is " << result2 << endl;

```

Output:

The number is positive
The number is positive

Example 6.14: finding the maximum of two numbers

```

1  int a = 15, b = 20;
2
3  // Using if-else statement
4  int max1;
5  if (a > b)
6      max1 = a;
7  else
8      max1 = b;
9
10  cout << "Maximum (if-else): " << max1 << endl;
11
12  // Using ternary operator (compact form)
13  int max2 = (a > b) ? a : b;
14
15  cout << "Maximum (ternary): " << max2 << endl;

```

Output:

Maximum (if-else): 20
Maximum (ternary): 20

Example 6.15: checking even or odd

```

1  int n = 7;
2
3  // Using ternary operator
4  string parity = (n % 2 == 0) ? "even" : "odd";
5
6  cout << n << " is " << parity << endl;

```

Output:

7 is odd

Example 6.16: assigning the absolute value

```
1 int x = -5;
2
3 // Using ternary operator to compute absolute value
4 int absValue = (x >= 0) ? x : -x;
5
6 cout << "Absolute value of " << x << " is " << absValue << endl;
```

Output:

Absolute value of -5 is 5

Example 6.17: nested ternary operators

```
1 int score = 85;
2
3 // Nested ternary operators to assign a grade
4 string grade = (score >= 90) ? "A" :
5                 (score >= 80) ? "B" :
6                 (score >= 70) ? "C" :
7                 (score >= 60) ? "D" : "F";
8
9 cout << "Score: " << score << ", Grade: " << grade << endl;
```

Output:

Score: 85, Grade: B

The conditions are evaluated from top to bottom: `score >= 90` is false and `score >= 80` is true, so the grade is "B". For readability, complex conditions are better handled with `if-else` statements.

6.6 Loops

Conditional statements choose *whether* a block of code is executed; loops decide *how many times* it is executed.

- Loops are used to execute a block of code repeatedly while a condition is met. They are essential for automating repetitive tasks and reducing redundant code. In C++, the most commonly used loops are the `while` loop and the `for` loop.

6.6.1 The while loop

- The **while** loop executes a block of code as long as the condition inside the parentheses () evaluates to **true**. When the condition is false, the loop terminates.

Example 6.18: while loop

```
1 int i = 1; // Initialization
2
3 while (i <= 5) // Condition
4 {
5     cout << "Number: " << i << endl;
6     i++; // Increment
7 }
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

The condition `i <= 5` is evaluated six times: five times it is true, and the sixth time (`i` equal to 6) it is false and the loop terminates.

6.6.2 The for loop

- The **for** loop is a compact loop structure that combines initialization, condition, and increment (or decrement) in one statement.

Example 6.19: for loop

```
1 for (int i = 1; i <= 5; i++) // Initialization, condition, increment
2 {
3     cout << "Number: " << i << endl;
4 }
```

The output is the same as in the **while** loop of Example 6.18.

6.6.3 The break and continue statements

- The **break** statement immediately terminates the loop.
- The **continue** statement skips the rest of the current iteration and moves to the next one.

Example 6.20: break and continue

```

1 for (int i = 1; i <= 10; i++)
2 {
3     if (i % 2 == 0)
4     {
5         continue; // Skip the even numbers
6     }
7     if (i > 7)
8     {
9         break; // Stop the loop at the first odd number greater than 7
10    }
11    cout << i << " ";
12 }
13 cout << endl;

```

Output:

1 3 5 7

The even values of *i* are skipped by **continue**; when *i* is equal to 9 the **break** statement terminates the loop, and the value 10 is never reached.

6.7 Arrays

Loops become really useful when they visit collections of values. A collection of variables of the same data type is stored in an **array**.

6.7.1 Fixed-size arrays

- An array whose values are known when the program is written can be declared by listing its values inside curly braces. The size of the array is the number of listed values, and it cannot change during the execution.
- Each element of an array is accessed using its **index**, starting from zero, inside square brackets “[]”: the elements of an array of size *n* are *b*[0], *b*[1], ..., *b*[*n*-1].

Example 6.21: fixed-size arrays

```

1 int b[] = {9, 12, 3, 16}; // Array of 4 integers
2 int n = 4;
3
4 b[2] = 5; // Assign a new value to the third element
5
6 for (int i = 0; i < n; i++)
7 {
8     cout << b[i] << " ";

```

```

9 }
10 cout << endl;

```

Output:

```

9 12 5 16

```

- A two-dimensional fixed-size array is declared with two sizes, the number of rows and the number of columns, and its values are listed row by row: `int c[2][3] = {{3, 1, 4}, {1, 5, 9}};` declares an array with 2 rows and 3 columns, and `c[1][2]` is the element in the second row and third column, equal to 9.

6.7.2 Dynamic-size arrays

- A **dynamic-size array** is a collection of variables of the same data type, stored in contiguous memory locations. The size of the array is determined during the execution of the algorithm.
- The reserved symbol “*” is used to declare a **pointer**, which stores the memory address of the first element of a dynamic-size array. The **new** operator is used to allocate the memory of a dynamic-size array, and it must specify the data type of the elements along with the size of the array inside square brackets “[]”.
- Each element of a dynamic-size array is accessed using its index, starting from zero, inside square brackets “[]”. A dynamic-size array requires memory deallocation using the `delete[]` operator to free memory and prevent memory leaks.

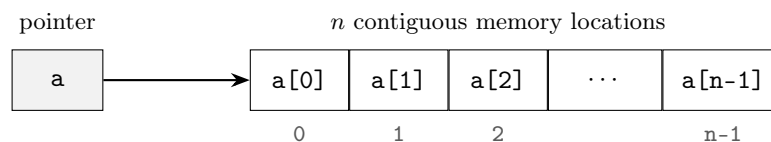


Figure 6.1: A dynamic-size array of size n : the pointer **a** stores the address of the first element, and the elements are indexed from 0 to $n - 1$.

Example 6.22: dynamic-size arrays

```

1 int n = 5; // Size of the array
2
3 int* a = new int[n]; // Memory allocation of an array of size n
4
5 for (int i = 0; i < n; i++)
6 {
7     a[i] = i * 2; // Example of value assignment
8 }
9
10 for (int i = 0; i < n; i++)
11 {
12     cout << a[i] << " ";
13 }
14 cout << endl;

```

```

15
16 delete[] a; // Free allocated memory

```

Output:

```
0 2 4 6 8
```

- Dynamic-size arrays can be multidimensional. For example, the reserved symbol “**” is used to declare a pointer to a pointer, which stores the memory address of the first element of a dynamic-size two-dimensional array: an array of m pointers, each pointing to a row of n elements.
- The `new` operator is also used to allocate the memory of dynamic-size two-dimensional arrays: first the array of the m row pointers, then each row. Each element of a dynamic-size two-dimensional array is accessed using two indices, both starting from zero, inside square brackets “[]”. The memory is freed in the reverse order: first each row, then the array of row pointers.

Example 6.23: dynamic-size two-dimensional arrays

```

1 int m = 3; // Number of rows
2 int n = 4; // Number of columns
3
4 int** matrix = new int*[m]; // Memory allocation of a matrix of size m times n
5 for (int i = 0; i < m; i++)
6 {
7     matrix[i] = new int[n]; // Allocate each row
8 }
9
10 for (int i = 0; i < m; i++)
11 {
12     for (int j = 0; j < n; j++)
13     {
14         matrix[i][j] = i * j; // Example of value assignment
15         cout << matrix[i][j] << " ";
16     }
17     cout << endl;
18 }
19
20 for (int i = 0; i < m; i++) // Free allocated memory
21 {
22     delete[] matrix[i]; // Free each row
23 }
24 delete[] matrix; // Free row pointers

```

Output:

```
0 0 0 0
0 1 2 3
0 2 4 6
```

6.8 The main and custom functions

As in pseudo-code, an algorithm is written as a function that receives the input and returns the output.

- Functions allow programmers to write modular and reusable code by grouping instructions into a single unit. A function can be called multiple times to execute the same task, reducing redundancy and improving code organization.
- Functions typically return a value and accept multiple parameters as input. When calling a function, the arguments are passed to the parameters.
- A function in C++ consists of:
 1. a **return type**: specifies the type of the value returned by the function (**void** if nothing is returned);
 2. a **function name**: the unique identifier of the function;
 3. a **parameter list** (optional): specifies the input values of the function;
 4. a **body**: contains the block of code executed when the function is called;
 5. a **return statement** (optional): used to return a value if the function returns a value.
- The **main** function serves as the entry point of a C++ program. It is executed first when the program starts running. Every C++ program must contain exactly one **main** function. Its return value 0 tells the operating system that the program terminated correctly.
- A function must be defined (or declared) before the point where it is called: for this reason the custom functions are written before **main**.
- C++ source code can also be compiled and executed online, for example at the following address: <https://www.mycompiler.io/it/online-c++-compiler>

Example 6.24: functions

```
1 #include <iostream>
2
3 using namespace std;
4
5 // Function definition
6 int addition(int a, int b)
7 {
8     return a + b; // Returns the sum
9 }
10
11 // Function definition
12 int multiplication(int a, int b)
13 {
14     return a * b; // Returns the product
15 }
16
17 int main()
```

```

18 {
19     int sum = addition(5, 10); // Function call
20     cout << "Sum: " << sum << endl;
21     int prod = multiplication(5, 10); // Function call
22     cout << "Prod: " << prod << endl;
23
24     return 0;
25 }

```

Output:

Sum: 15
Prod: 50

6.8.1 Parameters passed by value and by reference

A function returns at most one value. When a function has to produce more than one result, or has to modify a variable of the caller, its parameters are passed by reference.

- By default, a parameter is passed **by value**: the function receives a copy of the argument, and the changes made to the copy do not affect the variable of the caller.
- A parameter is passed **by reference** by writing the reserved symbol “&” after its data type (for example `int& x`): the parameter is another name for the variable of the caller, and every change made by the function is visible to the caller.
- An array is passed to a function through the pointer to its first element (for example `int *a`), together with its size: the function works directly on the elements of the array of the caller.

Example 6.25: parameters passed by value and by reference

```

1 #include <iostream>
2
3 using namespace std;
4
5 void incrementByValue(int x)
6 {
7     x = x + 1; // Modifies only the local copy
8 }
9
10 void incrementByReference(int& x)
11 {
12     x = x + 1; // Modifies the variable of the caller
13 }
14
15 void fillWithZeros(int n, int* a)
16 {
17     for (int i = 0; i < n; i++)
18     {
19         a[i] = 0; // Modifies the array of the caller
20     }
21 }

```

```

22
23 int main()
24 {
25     int k = 5;
26     incrementByValue(k);
27     cout << "After incrementByValue: " << k << endl;
28     incrementByReference(k);
29     cout << "After incrementByReference: " << k << endl;
30
31     int b[] = {4, 7, 1};
32     fillWithZeros(3, b);
33     cout << "Array: " << b[0] << " " << b[1] << " " << b[2] << endl;
34
35     return 0;
36 }

```

Output:

```

After incrementByValue: 5
After incrementByReference: 6
Array: 0 0 0

```

6.9 A complete algorithm: the maximum value in an array

We now put all the pieces together: a function, a loop, a conditional statement and a dynamic-size array.

We consider the problem of finding the maximum value in an array a of $n \geq 1$ integer numbers.

- This is the pseudo-code of the algorithm `FindMaximum` of the chapter «Pseudo-code»: the entries are indexed from 1 to n .

Algorithm: FindMaximum

Input : An array $a[1..n]$ of $n \geq 1$ integer numbers

Output: The maximum value in the array a

```

1 function FindMaximum( $a, n$ )
2    $max := a[1]$ 
3   for  $i := 2$  to  $n$  do
4     if  $a[i] > max$  then
5        $max \leftarrow a[i]$ 
6     end
7   end
8   return  $max$ 
9 end

```

- This is the C++ implementation: the entries are indexed from 0 to $n-1$, so the initialization uses $a[0]$ and the loop starts from $i = 1$. The function assumes $n \geq 1$, as the pseudo-code does: for $n = 0$ the element $a[0]$ does not exist.

```

1  #include <iostream>
2
3  using namespace std;
4
5  // Function to find the maximum value in an array
6  int ArrayMaxValue(int n, int *a)
7  {
8      int max = a[0]; // Initialize with the first element
9
10     for (int i = 1; i < n; i++)
11     {
12         if (a[i] > max)
13         {
14             max = a[i]; // Update the maximum
15         }
16     }
17
18     return max;
19 }
20
21 int main()
22 {
23     int n = 7; // Size of the array
24
25     int* a = new int[n]; // Memory allocation of an array of size n
26
27     int values[] = {9, 12, 3, 16, 19, 11, 12};
28     for (int i = 0; i < n; i++)
29     {
30         a[i] = values[i];
31     }
32
33     int max = ArrayMaxValue(n, a);
34     cout << "Maximum value: " << max << endl;
35
36     delete[] a; // Free allocated memory
37
38     return 0;
39 }

```

Output:

Maximum value: 19

Example 6.26: step-by-step execution

In each table, the column i is the index of the C++ loop and the column $i + 1$ is the corresponding index of the pseudo-code. The first row is the initialization $\text{max} = a[0]$.

Instance 1: $a = [9, 12, 3, 16, 19, 11, 12]$, $n = 7$

i (C++)	$i + 1$ (pseudo-code)	$a[i]$	$a[i] > \text{max?}$	max
—	—	—	—	9
1	2	12	Yes	12
2	3	3	No	12
3	4	16	Yes	16
4	5	19	Yes	19
5	6	11	No	19
6	7	12	No	19

Return: $\text{max} = 19$

Instance 2: $a = [3, 7, 2, 9, 1]$, $n = 5$

i (C++)	$i + 1$ (pseudo-code)	$a[i]$	$a[i] > \text{max?}$	max
—	—	—	—	3
1	2	7	Yes	7
2	3	2	No	7
3	4	9	Yes	9
4	5	1	No	9

Return: $\text{max} = 9$

6.10 Bitwise operators

The integer numbers handled by a program are stored in binary, as described in the chapter «Representation of numbers». C++ provides operators that act directly on these binary digits.

- **Bitwise operators** perform operations on the individual bits of integer values. They are essential for low-level programming, optimization, and working with binary representations.
- The bits are numbered from the right, starting from position 0 (the least significant bit).

Bitwise operators in C++:

- $\&$ (Bitwise AND) $\rightsquigarrow$ Returns 1 only if both corresponding bits are 1.
- $|$ (Bitwise OR) $\rightsquigarrow$ Returns 1 if at least one corresponding bit is 1.
- $\wedge$ (Bitwise XOR) $\rightsquigarrow$ Returns 1 if the corresponding bits are different.
- $\sim$ (Bitwise NOT) $\rightsquigarrow$ Flips all bits (0 becomes 1, 1 becomes 0).
- $<<$ (Left shift) $\rightsquigarrow$ Shifts the bits to the left by a specified number of positions.
- $>>$ (Right shift) $\rightsquigarrow$ Shifts the bits to the right by a specified number of positions.

6.10.1 Bitwise AND (&)

- The bitwise AND operator compares each bit of two numbers and returns 1 only if both bits are 1.
- Truth table for AND:

Bit A	Bit B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Example 6.27: bitwise AND

```

1 int a = 12; // Binary: 1100
2 int b = 10; // Binary: 1010
3 int result = a & b; // Binary: 1000 (decimal: 8)
4
5 cout << "a & b = " << result << endl; // Output: 8

```

Bit-by-bit operation:

		Decimal
a	1 1 0 0	12
b	1 0 1 0	10
a & b	1 0 0 0	8

6.10.2 Bitwise OR (|)

- The bitwise OR operator compares each bit of two numbers and returns 1 if at least one bit is 1.
- Truth table for OR:

Bit A	Bit B	A B
0	0	0
0	1	1
1	0	1
1	1	1

Example 6.28: bitwise OR

```

1 int a = 12; // Binary: 1100
2 int b = 10; // Binary: 1010

```

```

3 int result = a | b; // Binary: 1110 (decimal: 14)
4
5 cout << "a | b = " << result << endl; // Output: 14

```

Bit-by-bit operation:

		Decimal
a	1 1 0 0	12
b	1 0 1 0	10
a b	1 1 1 0	14

6.10.3 Bitwise XOR (^)

- The bitwise XOR (exclusive OR) operator compares each bit of two numbers and returns 1 if the bits are different.
- Truth table for XOR:

Bit A	Bit B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Example 6.29: bitwise XOR

```

1 int a = 12; // Binary: 1100
2 int b = 10; // Binary: 1010
3 int result = a ^ b; // Binary: 0110 (decimal: 6)
4
5 cout << "a ^ b = " << result << endl; // Output: 6

```

Bit-by-bit operation:

		Decimal
a	1 1 0 0	12
b	1 0 1 0	10
a ^ b	0 1 1 0	6

6.10.4 Bitwise NOT (~)

- The bitwise NOT operator flips all bits of a number (0 becomes 1, 1 becomes 0).
- This is a unary operator (it operates on a single operand).
- The result depends on the number of bits used to represent the integer. In the example we use the data type `unsigned char`, which stores the natural numbers from 0 to 255 with 8 bits; the conversion `(int)` displays the result as a number instead of a character.

Example 6.30: bitwise NOT

```

1 unsigned char a = 12; // Binary (8 bits): 00001100
2 unsigned char result = ~a; // Binary: 11110011 (decimal: 243)
3
4 cout << "a = " << (int)result << endl; // Output: 243

```

Bit-by-bit operation (using the 8-bit representation):

									Decimal
a	0	0	0	0	1	1	0	0	12
~a	1	1	1	1	0	0	1	1	243

Indeed, $243 = 255 - 12$: flipping all the 8 bits of a number x gives $255 - x$.

6.10.5 Left shift (<<)

- The left shift operator shifts all bits to the left by a specified number of positions.
- Zeros are added on the right side.
- Shifting left by k positions is equivalent to multiplying by 2^k (as long as the result can be represented by the data type).

Example 6.31: left shift

```

1 int a = 5; // Binary: 0101
2 int result = a << 2; // Shift left by 2 positions: 10100 (decimal: 20)
3
4 cout << "a << 2 = " << result << endl; // Output: 20
5 // Equivalent to: 5 * 2^2 = 5 * 4 = 20

```

Bit-by-bit operation:

		Decimal	Equivalent
a	0 0 0 1 0 1	5	5
a << 1	0 0 1 0 1 0	10	5×2^1
a << 2	0 1 0 1 0 0	20	5×2^2
a << 3	1 0 1 0 0 0	40	5×2^3

6.10.6 Right shift (>>)

- The right shift operator shifts all bits to the right by a specified number of positions; the bits that exit on the right are lost.
- For non-negative integers, zeros are added on the left side.
- For non-negative integers, shifting right by k positions is equivalent to the integer division by 2^k , that is, to $\lfloor x/2^k \rfloor$.

Example 6.32: right shift

```

1 int a = 20; // Binary: 10100
2 int result = a >> 2; // Shift right by 2 positions: 00101 (decimal: 5)
3
4 cout << "a >> 2 = " << result << endl; // Output: 5
5 // Equivalent to: 20 / 2^2 = 20 / 4 = 5

```

Bit-by-bit operation:

		Decimal	Equivalent
a	0 1 0 1 0 0	20	20
a >> 1	0 0 1 0 1 0	10	$20/2^1$
a >> 2	0 0 0 1 0 1	5	$20/2^2$
a >> 3	0 0 0 0 1 0	2	$\lfloor 20/2^3 \rfloor$

In the last row $20/2^3 = 2.5$, and the right shift gives the integer part 2: the bit 1 in position 0 of `a >> 2` is lost.

6.10.7 Practical applications

The bitwise operators give short and fast solutions to common tasks. In the following examples, a **mask** is a number whose binary representation has the bit 1 only in the positions we want to act on: the mask $1 \ll k$ has the bit 1 only in position k .

Example 6.33: checking if a number is even or odd

```
1 int n = 15;
2
3 if (n & 1)
4     cout << n << " is odd" << endl;
5 else
6     cout << n << " is even" << endl;
7
8 // Explanation: n & 1 checks the least significant bit
9 // If it is 1, the number is odd; if it is 0, the number is even
```

Output:

15 is odd

Example 6.34: swapping two numbers without a temporary variable

```
1 int a = 5, b = 9;
2
3 cout << "Before swap: a = " << a << ", b = " << b << endl;
4
5 a = a ^ b; // a now holds a XOR b
6 b = a ^ b; // b = (a XOR b) XOR b = a
7 a = a ^ b; // a = (a XOR b) XOR a = b
8
9 cout << "After swap: a = " << a << ", b = " << b << endl;
```

Output:

Before swap: a = 5, b = 9

After swap: a = 9, b = 5

The swap works because $x \oplus x$ is 0 and $x \oplus 0$ is x for every integer x , and the XOR is commutative and associative.

Example 6.35: checking if a specific bit is set

```
1 int n = 12; // Binary: 1100
2 int position = 2; // Check the bit at position 2 (from right, starting at 0)
3
4 if (n & (1 << position))
5     cout << "Bit " << position << " is set (1)" << endl;
6 else
7     cout << "Bit " << position << " is not set (0)" << endl;
8
```

```

9 // Explanation: (1 << position) creates a mask with only bit at position set
10 // n & mask checks if that bit is 1 in n

```

Output:

Bit 2 is set (1)

Example 6.36: setting a specific bit to 1

```

1 int n = 12; // Binary: 1100
2 int position = 1; // Set the bit at position 1
3
4 n = n | (1 << position); // Set bit at position 1 to 1
5
6 cout << "Result: " << n << endl; // Output: 14 (Binary: 1110)

```

Example 6.37: clearing a specific bit to 0

```

1 int n = 14; // Binary: 1110
2 int position = 1; // Clear the bit at position 1
3
4 n = n & ~(1 << position); // Set bit at position 1 to 0
5
6 cout << "Result: " << n << endl; // Output: 12 (Binary: 1100)

```

The mask $\sim(1 \ll \text{position})$ has the bit 0 in the given position and the bit 1 in all the other positions.

Example 6.38: toggling a specific bit

```

1 int n = 12; // Binary: 1100
2 int position = 0; // Toggle the bit at position 0
3
4 n = n ^ (1 << position); // Toggle bit at position 0
5
6 cout << "Result: " << n << endl; // Output: 13 (Binary: 1101)

```

Key points of the chapter

- A C++ program is compiled (for example with g++) and its execution starts from the main function; every instruction ends with “;”.
- Every variable is declared with its data type before use; the division between two int values is the integer division.
- Conditional statements (if, else, ?:) choose which instructions are executed; loops

(**while**, **for**) repeat them.

- The arrays of **C++** are indexed from 0 to $n - 1$, while the arrays of the pseudo-code are indexed from 1 to n ; a dynamic-size array is allocated with **new** and freed with **delete[]**.
- A function receives its parameters by value (a copy) or by reference (**&**); an array is passed through the pointer to its first element, together with its size.
- The bitwise operators act on the binary representation of the integer numbers: **x << k** multiplies x by 2^k and, for $x \geq 0$, **x >> k** computes $\lfloor x/2^k \rfloor$.

Introduction to Python

Introduction

In the chapter «Introduction to C++» we translated the pseudo-code of our algorithms into a compiled programming language. In this chapter we introduce a second programming language, **Python**, which is interpreted, has a lighter syntax and is very popular both in teaching and in scientific computing.

The chapter follows the same path as the chapter «Introduction to C++», so that the two languages can be compared section by section:

- the main characteristics of the language and the structure of a program;
- variables, data types, input and output;
- operators, conditional statements and loops;
- lists, which play the role of the arrays of C++;
- functions, their parameters and their return values;
- a complete algorithm, the search of the maximum value in an array, written in pseudo-code and in **Python**;
- the bitwise operators and a summary table that compares **Python** with C++.

Every fragment of code of the chapter has been executed with the **python3** interpreter (**Python 3.14**): the outputs reported in the examples are the ones printed by the programs.

In pseudo-code the entries of an array $a[1..n]$ are indexed from 1 to n ; in **Python**, as in C++, the elements of a list of size n are indexed from 0 to $n - 1$. The entry $a[i]$ of the pseudo-code corresponds to the element $a[i-1]$ of the **Python** code.

7.1 Main characteristics of the programming language

The main characteristics of Python are the following.

- Python is an **interpreted** language: programs are executed instruction by instruction by the **Python interpreter**, without a separate compilation step. This makes development and debugging fast and interactive.
- Python uses **dynamic typing**: variables do not need to be declared with a data type. The data type is determined automatically, during the execution, from the assigned value.
- Python supports **functions**, enabling a structured and modular approach to programming. Functions allow code reuse and better organization.
- **Indentation is mandatory** in Python: the block structure of the code (loops, conditional statements, functions) is defined by the level of indentation, not by curly braces. The standard indentation is 4 spaces.
- In Python, a single-line comment starts with the reserved symbol “#”. There is no dedicated symbol for multi-line comments: a text enclosed between triple quotes “""" (or “’”) is a string that is not used by the program, and it is commonly used as a multi-line comment. The same string written as the first line of a function is its **documentation string** (docstring).

Example 7.1: comments in Python

This is an example of a single-line comment:

```
1 # This is a comment line
```

This is an example of a multi-line comment:

```
1 """
2 This is a
3 multi-line comment
4 """
```

7.1.1 The structure of a program

A **Python** program is a text file (with extension `.py`) that is executed by the interpreter from the first line to the last one. Unlike **C++**, no `main` function is required: the instructions written at the top level of the file are executed in order.

Example 7.2: the skeleton of a program

```
1 # The instructions of the program are written here
2 print("Hello, world!")
```

If the source code is saved in the file `hello.py`, it is executed with the command

```
python3 hello.py
```

and the output is

```
Hello, world!
```

In the following examples, unless they contain a complete program, the fragments of code are written at the top level of a file and executed in this way.

- A **block** of instructions (the body of a conditional statement, of a loop or of a function) is introduced by the symbol “:” at the end of the preceding line, and all its instructions are indented by the same number of spaces. The block ends at the first instruction that is indented less.

Example 7.3: blocks defined by indentation

```
1 x = 7
2 if x > 0:
3     print("positive")           # Inside the block of the if
4     print("still inside")      # Inside the block of the if
5 print("outside the block")     # Executed in any case
```

Output:

```
positive
still inside
outside the block
```

In **C++** the same block would be enclosed in curly braces `{}`; in **Python** a wrong indentation changes the meaning of the program or produces an `IndentationError`.

In **Python** the instructions do not end with “;”, and the blocks are delimited by the indentation (4 spaces) instead of curly braces.

7.2 Variables and data types

The source code operates on **variables**, and every value has a **data type**.

- Information is categorized into different **data types**. The four fundamental data types are:
 1. The data type “**int**” represents an integer, which is a whole number written without a decimal point. This includes positive and negative whole numbers, as well as zero. Unlike C++, the integers of Python have no maximum value: they can have as many digits as needed.
 2. The data type “**bool**” represents a Boolean value: **True** or **False** (note the capital first letter).
 3. The data type “**float**” represents a floating-point number, which includes numbers written with a decimal point. A float of Python has the same (double) precision as a double of C++.
 4. The data type “**str**” represents a string of characters, enclosed in single or double quotes (e.g., ‘**hello**’, “**world**”). There is no separate data type for a single character: a character is a string of length 1.
- A variable acts as a container that stores a value. Each variable has a unique name.
- Variables are assigned values using the assignment operator “**=**”. When a new value is assigned, the previous value is overwritten.
- **No declaration is needed** before use: a variable is created the moment a value is assigned to it. The function `type()` returns the data type of a value.

Variables are created by assignment; there is no separate declaration step, and the data type is not written in the code.

Example 7.4: variable assignments

```

1 age = 25                # Integer variable
2 is_raining = False      # Boolean variable
3 small_number = 3.14     # Floating-point variable
4 large_number = 123456.789 # Floating-point variable
5 character = 'a'         # String variable (single character)
6 name = "Alice"          # String variable
7
8 age = 26 # The previous value 25 is overwritten
9 print(age, is_raining, small_number, large_number, character, name)
10 print(type(age), type(is_raining), type(small_number), type(name))

```

Output:

```

26 False 3.14 123456.789 a Alice
<class 'int'> <class 'bool'> <class 'float'> <class 'str'>

```

Compare with Example 7.6 below and with the corresponding example of the chapter «In-

roduction to C++»: the Boolean value is displayed as `False`, and the number `123456.789` is displayed with all its digits.

Example 7.5: dynamic typing and large integers

```

1 x = 5
2 print(x, type(x))
3 x = "five" # The same name now refers to a string
4 print(x, type(x))
5
6 print(2 ** 100) # An integer with 31 digits

```

Output:

```

5 <class 'int'>
five <class 'str'>
1267650600228229401496703205376

```

The operator `**` is the exponentiation (Section 7.4.1). In C++ the value 2^{100} does not fit in a variable of type `int`.

7.3 Input and output

- The function `print()` displays output on the screen. Multiple values can be printed in one call by separating them with commas: they are displayed separated by a space, and the line ends with a new line (the same effect as `endl` in C++).
- The optional parameters `sep` and `end` change the separator between the values and the text written at the end: `print(x, end=" ")` writes a space instead of a new line, so the next `print` continues on the same line.
- Strings and variable values can be combined using **f-strings** (formatted string literals), prefixed with the letter `f`. Inside f-strings, variable names (and, more generally, expressions) are placed between curly braces `{}`.
- Inside the curly braces of an f-string, the format `:.kf` displays a floating-point number with exactly k digits after the decimal point (the same effect as `fixed` and `setprecision(k)` in C++).

Example 7.6: screen output

```

1 n = 42
2 print("The value of the variable is", n)
3 print(f"The value of the variable is {n}") # f-string version
4 print(f"n = {n}, doubled = {n * 2}")      # Expression inside {}
5
6 print("a", "b", "c", sep="-")             # Separator "-" instead of a space
7 print("No new line", end=" ")             # A space instead of the new line

```

```

8 print("after end")
9 print(f"{10 / 3:.2f}")           # Two digits after the decimal point

```

Output:

```

The value of the variable is 42
The value of the variable is 42
n = 42, doubled = 84
a-b-c
No new line after end
3.33

```

- The function `input()` displays a message, waits for the user to type a line of text and returns it. It always returns a `str`; to obtain a number, the result must be converted with `int()` or `float()`.

Example 7.7: keyboard input

```

1 name = input("Enter your name: ")           # Reads a string
2 n = int(input("Enter an integer: "))         # Reads and converts to int
3 x = float(input("Enter a real number: "))    # Reads and converts to float
4 print(f"Hello {name}, n = {n}, x = {x}")

```

Example of execution (the user types Ada, 7 and 2.5):

```

Enter your name: Ada
Enter an integer: 7
Enter a real number: 2.5
Hello Ada, n = 7, x = 2.5

```

7.4 Operators

Operators are symbols that perform operations on variables and values. As in C++, they are classified, according to their functionality, into arithmetic, assignment, comparison and logical operators; Python has no increment and decrement operators.

7.4.1 Arithmetic operators

Arithmetic operators, used to perform basic mathematical operations:

- `+` (Addition) $\rightsquigarrow$ Adds two values.
- `-` (Subtraction) $\rightsquigarrow$ Subtracts one value from another.
- `*` (Multiplication) $\rightsquigarrow$ Multiplies two values.
- `/` (Division) $\rightsquigarrow$ Divides one value by another (always returns a `float`).
- `//` (Integer division) $\rightsquigarrow$ Returns the integer quotient (floor division).
- `%` (Modulus) $\rightsquigarrow$ Returns the remainder of a division.
- `**` (Exponentiation) $\rightsquigarrow$ Raises a value to a power.

Example 7.8: arithmetic operators

```

1 a = 10
2 b = 3
3
4 print("Addition:", a + b)
5 print("Subtraction:", a - b)
6 print("Multiplication:", a * b)
7 print("Division:", a / b)           # Always a float
8 print("Integer division:", a // b)
9 print("Modulus:", a % b)
10 print("Exponentiation:", a ** b)

```

Output:

```

Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Integer division: 3
Modulus: 1
Exponentiation: 1000

```

The last digit 5 of the division is not an error of the program: $10/3$ has infinitely many decimal digits, and the `float` closest to it is displayed.

- The division operator `/` always returns a `float` in Python 3, even when dividing two integers. To obtain an integer result, use the integer division operator `//`. This is the opposite convention with respect to C++, where $10 / 3$ is the integer division.
- The operator `//` computes $\lfloor a/b \rfloor$, rounding down. For non-negative operands it coincides with the integer division of C++; for a negative quotient it does not, because C++ truncates toward zero: in Python $-7 // 2$ is -4 and $-7 \% 2$ is 1 , while in C++ $-7 / 2$ is -3 and

$-7 \% 2$ is -1 . In both languages the identity $a = b \cdot q + r$ holds between quotient q and remainder r .

- A value is converted into another data type (**type conversion**) with the functions `int()`, `float()` and `str()`: `int(3.9)` is 3 (the decimal part is removed), `float(7)` is 7.0, `int("12")` is the integer 12.

Example 7.9: integer and floating-point division

```

1 a = 10
2 b = 3
3
4 print(a / b)           # Floating-point division
5 print(a // b)          # Integer division
6 print(f"{a / b:.2f}")  # Two decimals
7 print(-7 // 2, -7 % 2) # Floor division with a negative quotient
8 print(int(-7 / 2))      # Truncation toward zero, as in C++
9 print(int(3.9), float(7), int("12") + 1)
```

Output:

```

3.3333333333333335
3
3.33
-4 1
-3
3 7.0 13
```

7.4.2 Assignment operators

Assignment operators, used to assign values to variables:

- `=` (Assignment) $\rightsquigarrow$ Assigns a value to a variable.
- `+=` (Addition assignment) $\rightsquigarrow$ Adds a value and assigns the result.
- `-=` (Subtraction assignment) $\rightsquigarrow$ Subtracts a value and assigns the result.
- `*=` (Multiplication assignment) $\rightsquigarrow$ Multiplies and assigns the result.
- `/=` (Division assignment) $\rightsquigarrow$ Divides and assigns the result (a `float`).
- `//=` (Integer division assignment) $\rightsquigarrow$ Floor-divides and assigns the result.
- `%=` (Modulus assignment) $\rightsquigarrow$ Applies the modulus and assigns the result.
- `**=` (Exponentiation assignment) $\rightsquigarrow$ Raises to a power and assigns the result.

Example 7.10: assignment operators

```

1 x = 10  # Assignment
2 print("Initial value:", x)
3
4 x += 5   # Equivalent to: x = x + 5
5 print("After += 5:", x)
6
7 x -= 3   # Equivalent to: x = x - 3
8 print("After -= 3:", x)
9
10 x *= 2   # Equivalent to: x = x * 2
11 print("After *= 2:", x)
12
13 x //= 4   # Equivalent to: x = x // 4
14 print("After //= 4:", x)
15
16 x %= 4    # Equivalent to: x = x % 4
17 print("After %= 4:", x)
18
19 x **= 3   # Equivalent to: x = x ** 3
20 print("After **= 3:", x)
21
22 x /= 4    # Equivalent to: x = x / 4 (the result is a float)
23 print("After /= 4:", x)

```

Output:

```

Initial value: 10
After += 5: 15
After -= 3: 12
After *= 2: 24
After //= 4: 6
After %= 4: 2
After **= 3: 8
After /= 4: 2.0

```

7.4.3 Increment and decrement

- Python has no operators `++` and `--`: a variable is incremented with `x += 1` and decremented with `x -= 1`.
- The instruction `x++` is a syntax error. The expression `++x` is accepted, but it does not increment `x`: it applies twice the unary plus, so its value is `x` itself.

Example 7.11: increment and decrement

```

1 x = 5
2 x += 1      # Increment
3 print("After x += 1:", x)
4 x -= 1      # Decrement

```

```

5 print("After x -= 1:", x)
6 print("++x is", ++x, "and x is still", x)  # ++x is not an increment

```

Output:

```

After x += 1: 6
After x -= 1: 5
++x is 5 and x is still 5

```

7.4.4 Comparison operators

Comparison operators, used to compare values and return a Boolean result:

- `==` (Equal to) $\rightsquigarrow$ Checks if two values are equal.
 - `!=` (Not equal to) $\rightsquigarrow$ Checks if two values are different.
 - `<` (Less than) $\rightsquigarrow$ Checks if one value is smaller than another.
 - `>` (Greater than) $\rightsquigarrow$ Checks if one value is larger than another.
 - `<=` (Less than or equal to) $\rightsquigarrow$ Checks if one value is smaller than or equal to another.
 - `>=` (Greater than or equal to) $\rightsquigarrow$ Checks if one value is larger than or equal to another.
- Comparison operators return a Boolean result: **True** if the condition is satisfied, **False** otherwise (displayed as **True** and **False**, not as 1 and 0 as in C++). These operators are commonly used in **conditional statements** and **loops**.
 - Comparisons can be chained: `0 < x <= 10` means `0 < x` and `x <= 10`.

Example 7.12: comparison operators

```

1 a = 10
2 b = 5
3
4 print("a == b:", a == b)  # Equal to
5 print("a != b:", a != b)  # Not equal to
6 print("a < b:", a < b)    # Less than
7 print("a > b:", a > b)    # Greater than
8 print("a <= b:", a <= b)  # Less than or equal to
9 print("a >= b:", a >= b)  # Greater than or equal to
10 print("0 < b <= a:", 0 < b <= a)  # Chained comparison

```

Output:

```

a == b: False
a != b: True
a < b: False
a > b: True
a <= b: False
a >= b: True

```

```
0 < b <= a: True
```

7.4.5 Logical operators

Logical operators, used for logical operations, often in conditional statements:

- **and** (Logical AND) $\rightsquigarrow$ Returns **True** if both conditions are **True**.
 - **or** (Logical OR) $\rightsquigarrow$ Returns **True** if at least one condition is **True**.
 - **not** (Logical NOT) $\rightsquigarrow$ Reverses the Boolean value.
- The logical operators of Python are words, while in C++ they are the symbols `&&`, `||` and `!`.

Example 7.13: logical operators

```
1 condition1 = True
2 condition2 = False
3
4 print("condition1:", condition1)
5 print("condition2:", condition2)
6
7 # Logical AND: True only if both conditions are True
8 print("condition1 and condition2:", condition1 and condition2)
9
10 # Logical OR: True if at least one condition is True
11 print("condition1 or condition2:", condition1 or condition2)
12
13 # Logical NOT: reverses the Boolean value
14 print("not condition1:", not condition1)
15 print("not condition2:", not condition2)
```

Output:

```
condition1: True
condition2: False
condition1 and condition2: False
condition1 or condition2: True
not condition1: False
not condition2: True
```

7.5 Conditional statements

The comparison and logical operators produce the conditions that decide which instructions are executed.

- Conditional statements allow executing different blocks of code based on specific conditions.
- The `if` statement evaluates a condition. If the condition is `True`, the indented block is executed.
- The optional `elif` (else if) clause checks an additional condition when the previous ones are `False`.
- The optional `else` clause is executed when all preceding conditions are `False`.
- **No parentheses are needed** around the condition, and **no curly braces** are used: the line of the condition ends with `:` and the block is delimited by indentation.

Example 7.14: conditional statement

```

1 number = 10
2
3 # Check if number is positive
4 if number > 0:
5     print("The number is positive.")
6 else:
7     print("The number is non-positive.")

```

Output:

The number is positive.

Example 7.15: if / elif / else

```

1 score = 85
2
3 if score >= 90:
4     grade = "A"
5 elif score >= 80:
6     grade = "B"
7 elif score >= 70:
8     grade = "C"
9 elif score >= 60:
10    grade = "D"
11 else:
12    grade = "F"
13
14 print(f"Score: {score}, Grade: {grade}")

```

Output:

Score: 85, Grade: B

The conditions are evaluated from top to bottom, and only the block of the first true condition is executed: `score >= 90` is false and `score >= 80` is true, so the grade is "B". This is the same result as the nested ternary operators of the chapter «Introduction to C++».

7.5.1 Conditional expression

- The **conditional expression** is the Python version of the ternary operator `?:` of C++: it provides a compact way to write simple if-else statements.
- Syntax: `value_if_true if condition else value_if_false`
- If the condition is True, the expression evaluates to `value_if_true`; otherwise, it evaluates to `value_if_false`.

Example 7.16: conditional expression

```

1 number = 10
2
3 # Compact conditional expression
4 result = "positive" if number > 0 else "non-positive"
5 print(f"The number is {result}")
6
7 # Finding the maximum of two numbers
8 a, b = 15, 20 # Multiple assignment: a = 15 and b = 20
9 maximum = a if a > b else b
10 print(f"Maximum: {maximum}")
11
12 # Checking even or odd
13 n = 7
14 parity = "even" if n % 2 == 0 else "odd"
15 print(f"{n} is {parity}")
16
17 # Absolute value
18 x = -5
19 abs_value = x if x >= 0 else -x
20 print(f"Absolute value of {x} is {abs_value}")

```

Output:

The number is positive

Maximum: 20

7 is odd

Absolute value of -5 is 5

The Python library also provides the functions `max(a, b)` and `abs(x)`, which return the same values.

7.6 Loops

Conditional statements choose *whether* a block of code is executed; loops decide *how many times* it is executed.

- Loops are used to execute a block of code repeatedly. In **Python**, the two loop constructs are the **while** loop and the **for** loop.

7.6.1 The while loop

- The **while** loop executes a block of code as long as the condition evaluates to **True**. If the condition is **False** from the beginning, the block is never executed.

Example 7.17: while loop

```

1 i = 1                # Initialization
2
3 while i <= 5:        # Condition
4     print(f"Number: {i}")
5     i += 1           # Increment (there is no ++ operator in Python)

```

Output:

```

Number: 1
Number: 2
Number: 3
Number: 4
Number: 5

```

As in C++, the condition `i <= 5` is evaluated six times: five times it is true, and the sixth time (`i` equal to 6) it is false and the loop terminates.

7.6.2 The for loop

- The **for** loop of **Python** iterates over the elements of an **iterable**, such as a list, a string, or a range of integers: at each iteration the loop variable takes the next element.
- The function `range(start, stop, step)` generates the sequence of integers from **start** (included) to **stop** (excluded), with increment **step**. If only one argument is given, the sequence starts at 0; if **step** is omitted, it is 1; with a negative **step** the sequence decreases.
- The pseudo-code loop “**for** $i := 1$ **to** n ” corresponds to `for i in range(1, n + 1)`, and the loop over the indices of a list of size n is `for i in range(n)`, with $i = 0, 1, \dots, n - 1$.

Example 7.18: for loop with range

```

1 for i in range(1, 6): # i takes the values 1, 2, 3, 4, 5
2     print(f"Number: {i}")

```

The output is the same as in the **while** loop of Example 7.17. The function `list()` shows the integers generated by a **range**:

```

1 print(list(range(5)))
2 print(list(range(1, 6)))
3 print(list(range(0, 10, 3)))
4 print(list(range(5, 0, -1)))

```

Output:

```

[0, 1, 2, 3, 4]
[1, 2, 3, 4, 5]
[0, 3, 6, 9]
[5, 4, 3, 2, 1]

```

Example 7.19: for loop over a list

```

1 a = [3, 7, 2, 9, 5]
2
3 for element in a:
4     print(element)

```

Output:

```

3
7
2
9
5

```

The loop visits the elements directly, without using their indices.

7.6.3 The break and continue statements

- The **break** statement immediately terminates the loop.
- The **continue** statement skips the rest of the current iteration and moves to the next one.

Example 7.20: break and continue

```

1 for i in range(1, 11):
2     if i % 2 == 0:
3         continue # Skip the even numbers
4     if i > 7:
5         break # Stop the loop at the first odd number greater than 7
6     print(i, end=" ")
7 print()

```

Output:

```
1 3 5 7
```

As in the C++ version, the even values of *i* are skipped by **continue**; when *i* is equal to 9 the **break** statement terminates the loop, and the value 10 is never reached. The final **print()** writes only the new line.

7.7 Lists

Loops become really useful when they visit collections of values. In **Python** a collection of values is stored in a **list**.

- A **list** is an ordered, mutable collection of elements, that can hold values of any type. Lists are the main sequence data structure in **Python** and play the role of the arrays of **C++**.
- Lists are enclosed in square brackets `[]` and elements are separated by commas. Elements are accessed by their **index**, starting from 0: the elements of a list `a` of size n are `a[0]`, `a[1]`, ..., `a[n-1]`. The function `len(a)` returns the size of the list, and the negative index `a[-1]` denotes the last element.
- Unlike the arrays of **C++**, lists are **dynamic**: they can grow or shrink at runtime without manual memory management. The instruction `a.append(x)` adds the element `x` at the end of the list.
- The expression `[0] * n` creates a list of n zeros. More generally, a **list comprehension** `[expression for i in range(n)]` creates the list of the n values of the expression for $i = 0, 1, \dots, n-1$; when the loop variable is not used, it is conventionally called `_`.

Lists are created with `[]` and indexed starting from 0. No memory allocation or deallocation is required.

Example 7.21: creating and accessing a list

```

1  n = 5
2
3  # Create a list of n zeros
4  a = [0] * n
5
6  # Assign values
7  for i in range(n):
8      a[i] = i * 2
9
10 print(a)          # The whole list
11 print(a[0])       # First element
12 print(a[4])       # Last element, index n-1
13 print(len(a))     # Number of elements

```

Output:

```

[0, 2, 4, 6, 8]
0
8
5

```

Example 7.22: list literal, modification and growth

```
1 b = [9, 12, 3, 16]    # List of 4 integers
2
3 b[2] = 5              # Assign a new value to the third element
4 b.append(20)          # The list grows: now it has 5 elements
5 print(b, len(b))
6 print(b[-1])          # The last element
7
8 squares = [i * i for i in range(6)] # List comprehension
9 print(squares)
```

Output:

```
[9, 12, 5, 16, 20] 5
20
[0, 1, 4, 9, 16, 25]
```

Accessing an index that does not exist, for example `b[5]`, stops the program with the error `IndexError: list index out of range` (in C++ the same access is not checked).

Example 7.23: list literal and iteration over the indices

```
1 a = [9, 12, 3, 16, 19, 11, 12]
2 n = len(a)
3
4 for i in range(n):
5     print(f"a[{i}] = {a[i]}")
```

Output:

```
a[0] = 9
a[1] = 12
a[2] = 3
a[3] = 16
a[4] = 19
a[5] = 11
a[6] = 12
```

7.7.1 Two-dimensional lists

- Two-dimensional data can be represented as a **list of lists**. Each row is itself a list, and elements are accessed with two indices `matrix[i][j]`, both starting from 0: for example, if `c = [[3, 1, 4], [1, 5, 9]]`, then `c[1][2]` is the element in the second row and third column, equal to 9.
- A matrix of m rows and n columns filled with zeros is created with the list comprehension `[[0] * n for _ in range(m)]`, which builds m distinct rows.

Example 7.24: two-dimensional list (matrix)

```

1 m = 3      # Number of rows
2 n = 2      # Number of columns
3
4 # Create a matrix of zeros
5 matrix = [[0] * n for _ in range(m)]
6
7 # Assign values
8 for i in range(m):
9     for j in range(n):
10        matrix[i][j] = i * j
11
12 # Print the matrix
13 for i in range(m):
14     for j in range(n):
15        print(f"matrix[{i}][{j}] = {matrix[i][j]}")
16 print(matrix)

```

Output:

```

matrix[0][0] = 0
matrix[0][1] = 0
matrix[1][0] = 0
matrix[1][1] = 1
matrix[2][0] = 0
matrix[2][1] = 2
[[0, 0], [0, 1], [0, 2]]

```

- The shorter expression `[[0] * n] * m` is wrong: it repeats m times the *same* row, so a change to one row appears in all the rows.

Example 7.25: a common mistake with two-dimensional lists

```

1 wrong = [[0] * 2] * 3      # Three references to the same row
2 wrong[0][0] = 1
3 print(wrong)
4
5 right = [[0] * 2 for _ in range(3)] # Three distinct rows
6 right[0][0] = 1

```

```
7 print(right)
```

Output:

```
[[1, 0], [1, 0], [1, 0]]
[[1, 0], [0, 0], [0, 0]]
```

7.8 Functions

As in pseudo-code, an algorithm is written as a function that receives the input and returns the output.

- Functions allow programmers to write modular and reusable code by grouping instructions into a single unit. A function is defined with the keyword **def**, followed by the function name and the parameter list in parentheses, ending with the symbol “:”.
- A function in Python consists of:
 1. the keyword **def** followed by the **function name**;
 2. a **parameter list** (optional): specifies the input values, without their data types;
 3. an indented **body**: the block of code executed when the function is called;
 4. a **return** statement (optional): returns a value to the caller. A function without **return** returns the special value **None**.
- A function must be defined before the instruction that calls it is executed: for this reason the functions are written at the beginning of the file.
- The main instructions of a Python program are typically placed at the top level of the file, or inside a block guarded by `if __name__ == "__main__":`, which ensures that the code runs only when the file is executed directly (and not when it is imported as a module by another file). This block plays the role of the **main** function of C++.
- Python source code can also be executed online, for example at the following address:
<https://www.mycompiler.io/online-python-compiler>

Example 7.26: functions

```
1 # Function definition
2 def addition(a, b):
3     return a + b # Returns the sum
4
5 # Function definition
6 def multiplication(a, b):
7     return a * b # Returns the product
8
9 if __name__ == "__main__":
10     total = addition(5, 10) # Function call
11     print(f"Sum: {total}")
```

```

12
13     product = multiplication(5, 10)  # Function call
14     print(f"Prod: {product}")

```

Output:

Sum: 15
Prod: 50

Example 7.27: function operating on a two-dimensional list

```

1 def sum_matrix(matrix, rows, columns):
2     """Return the sum of all elements of a 2D list."""
3     total = 0
4     for i in range(rows):
5         for j in range(columns):
6             total += matrix[i][j]
7     return total
8
9 if __name__ == "__main__":
10     m, n = 3, 2
11
12     # Build the matrix with a nested list comprehension
13     matrix = [[i * j for j in range(n)] for i in range(m)]
14
15     result = sum_matrix(matrix, m, n)
16     print(f"Sum of matrix elements: {result}")

```

Output:

Sum of matrix elements: 3

The matrix is $\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 2 \end{bmatrix}$, as in Example 7.24, and the sum of its elements is $0 + 0 + 0 + 1 + 0 + 2 = 3$.

7.8.1 Parameters and return values

In C++ the parameters are passed by value or by reference. In Python every parameter receives the *same* object as the argument of the caller, and the effect of a function on the caller depends on what the function does with it.

- If the function **assigns a new value** to a parameter (for example $x = x + 1$), only the local name changes: the variable of the caller is not affected. For numbers, the behaviour is the same as the parameters passed by value in C++.
- If the function **modifies the elements** of a list received as a parameter (for example $a[i] = 0$), the change is visible to the caller, as for the arrays passed to a function in C++.
- A function can return **several values** at once, separated by commas: they are returned as a **tuple**, and the caller assigns them to several variables with a multiple assignment.

In C++ the same result requires parameters passed by reference.

Example 7.28: parameters and multiple return values

```

1 def increment(x):
2     x = x + 1          # Changes only the local name x
3
4 def fill_with_zeros(a):
5     for i in range(len(a)):
6         a[i] = 0      # Modifies the list of the caller
7
8 def divide(n, m):
9     return n // m, n % m # Returns two values (a tuple)
10
11 if __name__ == "__main__":
12     k = 5
13     increment(k)
14     print("After increment:", k)
15
16     b = [4, 7, 1]
17     fill_with_zeros(b)
18     print("List:", b)
19
20     q, r = divide(17, 5)
21     print("Quotient and remainder:", q, r)
22     print("Tuple:", divide(17, 5))

```

Output:

```

After increment: 5
List: [0, 0, 0]
Quotient and remainder: 3 2
Tuple: (3, 2)

```

7.9 A complete algorithm: the maximum value in an array

We now put all the pieces together: a function, a loop, a conditional statement and a list.

We consider the problem of finding the maximum value in an array a of $n \geq 1$ integer numbers.

- This is the pseudo-code of the algorithm `FindMaximum` of the chapter «Pseudo-code»: the entries are indexed from 1 to n .

Algorithm: FindMaximum**Input** : An array $a[1..n]$ of $n \geq 1$ integer numbers**Output**: The maximum value in the array a

```

1 function FindMaximum( $a, n$ )
2    $max := a[1]$ 
3   for  $i := 2$  to  $n$  do
4     if  $a[i] > max$  then
5        $max \leftarrow a[i]$ 
6     end
7   end
8   return  $max$ 
9 end

```

- This is the Python implementation. The elements are indexed from 0 to $n-1$, so the initialization uses $a[0]$ and the loop `for i in range(1, len(a))` visits the indices $1, \dots, n-1$. The size n is not passed as a parameter, because the function obtains it with `len(a)`. The variable is called `max_val` and not `max`, so as not to hide the built-in function `max()`. The function assumes $n \geq 1$, as the pseudo-code does: for an empty list the access to $a[0]$ stops the program with an `IndexError`.

```

1 def array_max_value(a):
2     """Return the maximum value in the list a (len(a) >= 1)."""
3     max_val = a[0] # Initialize with the first element
4
5     for i in range(1, len(a)):
6         if a[i] > max_val:
7             max_val = a[i] # Update the maximum
8
9     return max_val
10
11 if __name__ == "__main__":
12     a = [9, 12, 3, 16, 19, 11, 12]
13
14     max_val = array_max_value(a)
15     print(f"Maximum value: {max_val}")

```

Output:

Maximum value: 19

- The test instance can also be generated at random with the module `random` of the Python library: `random.randint(1, 100)` returns a random integer between 1 and 100 (both included). The instruction `random.seed(2026)` fixes the starting point of the random generator, so that every execution produces the same list; without it, every execution produces a different list.

Example 7.29: a random instance

```

1 import random
2
3 def array_max_value(a):
4     max_val = a[0]
5     for i in range(1, len(a)):
6         if a[i] > max_val:
7             max_val = a[i]
8     return max_val
9
10 if __name__ == "__main__":
11     random.seed(2026) # Same random list at every execution
12     n = 10             # Size of the list
13     a = [random.randint(1, 100) for _ in range(n)]
14
15     print(a)
16     print(f"Maximum value: {array_max_value(a)}")

```

Output:

```

[16, 41, 65, 66, 83, 14, 29, 77, 80, 72]
Maximum value: 83

```

Example 7.30: step-by-step execution

In each table, the column i is the index of the Python loop and the column $i + 1$ is the corresponding index of the pseudo-code. The first row is the initialization $\text{max_val} = a[0]$.

Instance 1: $a = [9, 12, 3, 16, 19, 11, 12]$, $n = 7$

i (Python)	$i + 1$ (pseudo-code)	$a[i]$	$a[i] > \text{max_val}?$	max_val
—	—	—	—	9
1	2	12	Yes	12
2	3	3	No	12
3	4	16	Yes	16
4	5	19	Yes	19
5	6	11	No	19
6	7	12	No	19

Return: $\text{max_val} = 19$

Instance 2: $a = [3, 7, 2, 9, 1]$, $n = 5$

i (Python)	$i + 1$ (pseudo-code)	$a[i]$	$a[i] > \text{max_val}?$	max_val
—	—	—	—	3
1	2	7	Yes	7
2	3	2	No	7
3	4	9	Yes	9
4	5	1	No	9

Return: `max_val = 9`

7.10 Bitwise operators

The bitwise operators of the chapter «Introduction to C++» exist in `Python` with the same symbols and the same meaning: `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (left shift) and `>>` (right shift). The function `bin()` returns the binary representation of an integer, preceded by the prefix `0b`.

- Since the integers of `Python` have no fixed number of bits, the bitwise NOT does not depend on a data type: `~x` is always equal to $-x - 1$. To obtain the 8-bit result of C++, the result is combined with the mask `0xFF` (the number 255, whose binary representation is `11111111`).

Example 7.31: bitwise operators

```

1 a = 12  # Binary: 1100
2 b = 10  # Binary: 1010
3
4 print(a & b, a | b, a ^ b)           # AND, OR, XOR
5 print(bin(a), bin(b), bin(a & b))   # Binary representations
6 print(5 << 2, 20 >> 2, 20 >> 3)      # Shifts
7 print(~a, ~a & 0xFF)                # NOT, and its 8-bit version
8
9 n = 15
10 print("odd" if n & 1 else "even")   # Least significant bit

```

Output:

```

8 14 6
0b1100 0b1010 0b1000
20 5 2
-13 243
odd

```

All the values coincide with those of the corresponding examples of the chapter «Introduction to C++».

7.11 Python and C++ side by side

The following table summarizes the main differences between the two languages met in this chapter.

	C++	Python
Execution	compiled (g++), then executed	interpreted (python3)
Data types	declared: <code>int x = 5;</code>	not declared: <code>x = 5</code>
End of instruction	<code>;</code>	end of the line
Blocks	curly braces <code>{}</code>	<code>:</code> and indentation
Comments	<code>//</code> and <code>/* */</code>	<code>#</code> (and <code>""" """</code>)
Boolean values	<code>true</code> , <code>false</code> (printed 1, 0)	<code>True</code> , <code>False</code>
<code>10 / 3</code>	3 (integer division)	3.3333333333333335
Integer division	<code>/</code> between two <code>int</code>	<code>//</code>
Power	function <code>pow</code>	operator <code>**</code>
Increment	<code>x++</code> , <code>++x</code>	<code>x += 1</code>
Logical operators	<code>&&</code> , <code> </code> , <code>!</code>	<code>and</code> , <code>or</code> , <code>not</code>
Ternary operator	<code>c ? x : y</code>	<code>x if c else y</code>
for loop	<code>for (int i = 0; i < n; i++)</code>	<code>for i in range(n):</code>
Collections	arrays, fixed or dynamic size	lists, dynamic
Size of a collection	stored in a separate variable	<code>len(a)</code>
Several results	parameters by reference	return of a tuple
Entry point	function <code>main</code>	top level of the file

Key points of the chapter

- A Python program is executed by the interpreter (python3) from the first line to the last one; the instructions do not end with “;” and the blocks are delimited by “:” and by the indentation.
- Variables are created by assignment, without declaring their data type; the integers have no maximum value.
- The operator `/` always gives a `float`; the integer division is `//` and it rounds down, also for negative numbers.
- Conditional statements (`if`, `elif`, `else`, and the conditional expression) choose which instructions are executed; loops (`while`, and `for` over a `range` or a list) repeat them.
- The lists of Python are indexed from 0 to $n - 1$, with $n = \text{len}(a)$, while the arrays of the pseudo-code are indexed from 1 to n ; lists grow with `append` and need no memory management.
- A function modifies the list of the caller when it changes its elements, but not a number it receives; several results are returned together as a tuple.

Introduction to Java

Introduction

In the chapters «Introduction to C++» and «Introduction to Python» we translated the pseudo-code of our algorithms into a compiled and into an interpreted programming language. In this chapter we introduce a third programming language, **Java**, which is in between: its programs are compiled into an intermediate code, which is then executed by a virtual machine. Its syntax is very close to the syntax of C++, but every program is organized into **classes**.

The chapter follows the same path as the two previous chapters, so that the three languages can be compared section by section:

- the main characteristics of the language and the structure of a program;
- variables, data types, input and output;
- operators, conditional statements and loops;
- arrays, and the dynamic lists of the class `ArrayList`;
- methods (the **Java** name for functions), their parameters and their return values;
- a complete algorithm, the search of the maximum value in an array, written in pseudo-code and in **Java**;
- the bitwise operators and a summary table that compares **Java** with C++ and Python.

Every fragment of code of the chapter has been compiled with the `javac` compiler and executed with the `java` virtual machine (OpenJDK 25): the outputs reported in the examples are the ones printed by the programs.

In pseudo-code the entries of an array $a[1..n]$ are indexed from 1 to n ; in **Java**, as in C++ and Python, the elements of an array of size n are indexed from 0 to $n - 1$. The entry $a[i]$ of the pseudo-code corresponds to the element `a[i-1]` of the Java code.

8.1 Main characteristics of the programming language

The main characteristics of **Java** are the following.

- **Java** is a **compiled and interpreted** language: the source code (`.java` files) is first compiled into **bytecode** (`.class` files) by the **Java** compiler `javac`, and then the bytecode is executed by the **Java Virtual Machine** (JVM). This makes **Java** programs **platform-independent**: the same bytecode runs on any machine that has a JVM installed (“write once, run anywhere”).
- **Java** is a **strongly and statically typed** language: every variable must be declared with its data type before use, and its data type cannot change during the execution.
- **Java** is **object-oriented**: every program is organized into **classes**. The entry point of every **Java** program is the `main` method of a class.
- **Java** supports **methods** (the **Java** term for functions), enabling a modular and reusable approach to programming.
- As in **C++**, the block structure of the code is defined by **curly braces** `{}`, and each instruction must end with the reserved symbol `“;”`.
- As in **C++**, a single-line comment starts with the reserved symbol `“//”`, and multi-line comments are enclosed between the reserved symbols `“/*”` and `“*/”`.

Example 8.1: comments in Java

This is an example of a single-line comment:

```
1 // This is a comment line
```

This is an example of a multi-line comment:

```
1 /* This is a  
2    multi-line comment */
```

8.1.1 The structure of a program

Before describing the single instructions, we look at the skeleton that every program shares. A Java program is made of one or more classes; the execution starts from the method `public static void main(String[] args)` of the class that is launched.

Example 8.2: the skeleton of a program

```

1 public class HelloWorld {
2
3     public static void main(String[] args) {
4         // The instructions of the program are written here
5         System.out.println("Hello, world!");
6     }
7 }

```

A class declared `public` must be saved in a file with the same name as the class: here `HelloWorld.java`. The program is compiled and executed with the commands

```

javac HelloWorld.java
java HelloWorld

```

The first command produces the bytecode file `HelloWorld.class`; the second one executes it (note that the extension is not written). The output is

```
Hello, world!
```

In the following examples we write only the instructions that matter: unless they contain a complete program, the fragments of code are meant to be placed inside the curly braces of `main`.

Every Java instruction belongs to a method, and every method belongs to a class. Each instruction ends with “;”.

8.2 Variables and data types

The source code operates on **variables**, and every variable has a **data type**.

- The four fundamental data types of Java are:
 1. The data type “`int`” represents an integer stored with 32 bits: a whole number from $-2^{31} = -2147483648$ to $2^{31} - 1 = 2147483647$. For larger integers the data type `long` (64 bits) is used.
 2. The data type “`boolean`” represents a Boolean value: `true` or `false` (lowercase).
 3. The data type “`double`” represents a floating-point number with double precision (64 bits), as in C++. The data type `float` has single precision, and its values are written with the suffix `f` (e.g., `3.14f`).
 4. The data type “`char`” represents a single character, enclosed in single quotes (e.g., `'A'`, `'7'`).

- The data type “**String**” (with capital S) represents a sequence of characters enclosed in double quotes, and it is the standard way to handle text in Java.
- A variable acts as a container that stores a value of a specific data type for later use.
- Variables are assigned values using the assignment operator “=”. When a new value is assigned, the previous value is overwritten.
- The compiler checks the data types: assigning a value of a wrong type (for example the string “five” to a variable of type `int`) or using a variable that has not received a value is a **compilation error**, and the program is not produced.

Variables must be declared with their data type before use, as in C++.

Example 8.3: variable declarations and assignments

```

1 int age = 25;           // Integer variable
2 boolean isRaining = false; // Boolean variable
3 double pi = 3.14159;     // Double-precision floating-point
4 float small = 3.14f;     // Single-precision floating-point
5 double large = 123456.789; // Double-precision floating-point
6 char letter = 'A';      // Character variable
7 String name = "Alice";  // String variable
8
9 age = 26; // The previous value 25 is overwritten
10 System.out.println(age + " " + isRaining + " " + pi + " " + small);
11 System.out.println(large + " " + letter + " " + name);

```

Output:

```

26 false 3.14159 3.14
123456.789 A Alice

```

Unlike C++, the Boolean value is displayed as `false` and the number 123456.789 is displayed with all its digits.

Example 8.4: the limits of the data type `int`

```

1 int big = Integer.MAX_VALUE; // The largest int
2 System.out.println(big);
3 System.out.println(big + 1); // Overflow: the result does not fit in 32 bits
4
5 long bigger = (long) big + 1; // Computed with 64 bits
6 System.out.println(bigger);

```

Output:

```

2147483647
-2147483648
2147483648

```

When the result of an operation between `int` values does not fit in 32 bits, it “wraps around”

to the other end of the interval, without any error message. The integers of Python never overflow.

8.3 Input and output

- The object `System.out` (the standard output, provided by the class `System`) has the methods that display values on the screen. The most common are:
 - `System.out.println()`: prints a value followed by a new line;
 - `System.out.print()`: prints a value without the new line.
- Values and strings are **concatenated** with the `+` operator inside a print statement: when one of the two operands of `+` is a `String`, the other one is converted into text. The expression is evaluated from left to right, so an arithmetic operation inside a concatenation must be enclosed in parentheses.
- The method `System.out.printf()` prints a formatted text: in the first string, `%d` is replaced by an integer, `%s` by a string, `%.kf` by a floating-point number with exactly k digits after the decimal point, and `%n` by a new line.

Example 8.5: screen output

```

1 int n = 42;
2 System.out.println("The value of the variable is " + n);
3 System.out.println("n = " + n + ", doubled = " + (n * 2));
4
5 System.out.println("n + 1 = " + n + 1); // Concatenation: "42" then "1"
6 System.out.println("n + 1 = " + (n + 1)); // Addition, then concatenation
7
8 System.out.print("No new line ");
9 System.out.println("after print");
10 System.out.printf("%d divided by %d is %.2f%n", 10, 3, 10.0 / 3);

```

Output:

```

The value of the variable is 42
n = 42, doubled = 84
n + 1 = 421
n + 1 = 43
No new line after print
10 divided by 3 is 3.33

```

- The decimal separator printed by `printf` depends on the language settings of the computer: on a computer set to Italian the last line is 10 divided by 3 is 3,33, with a comma. The method `println` always uses the point.
- To read input from the keyboard, the class `Scanner` of the package `java.util` is used. It must be imported with the instruction `import java.util.Scanner;`, written at the beginning of the file, before the class. The methods `nextLine()`, `nextInt()` and `nextDouble()` read a line of text, an integer and a floating-point number.

Example 8.6: keyboard input with Scanner

```

1  import java.util.Scanner;
2
3  public class InputExample {
4
5      public static void main(String[] args) {
6          Scanner sc = new Scanner(System.in);
7
8          System.out.print("Enter your name: ");
9          String name = sc.nextLine();    // Reads a String
10
11         System.out.print("Enter an integer: ");
12         int n = sc.nextInt();           // Reads an int
13
14         System.out.print("Enter a real number: ");
15         double x = sc.nextDouble();     // Reads a double
16
17         System.out.println("Hello " + name + ", n = " + n + ", x = " + x);
18         sc.close();
19     }
20 }

```

Example of execution (the user types Ada, 7 and 2.5):

```

Enter your name: Ada
Enter an integer: 7
Enter a real number: 2.5
Hello Ada, n = 7, x = 2.5

```

Like `printf`, the method `nextDouble()` follows the language settings: on a computer set to Italian the real number must be typed as 2,5.

8.4 Operators

Operators are symbols that perform operations on variables and values. The operators of Java are the same as those of C++: arithmetic, assignment, increment and decrement, relational and logical operators.

8.4.1 Arithmetic operators

Arithmetic operators, used to perform basic mathematical operations:

- `+` (Addition) $\rightsquigarrow$ Adds two values.
- `-` (Subtraction) $\rightsquigarrow$ Subtracts one value from another.
- `*` (Multiplication) $\rightsquigarrow$ Multiplies two values.
- `/` (Division) $\rightsquigarrow$ Divides one value by another. Integer division when both operands are integers.
- `%` (Modulus) $\rightsquigarrow$ Returns the remainder of a division.

Example 8.7: arithmetic operators

```

1 int a = 10, b = 3;
2
3 System.out.println("Addition:      " + (a + b));
4 System.out.println("Subtraction:   " + (a - b));
5 System.out.println("Multiplication: " + (a * b));
6 System.out.println("Division:      " + (a / b)); // Integer division
7 System.out.println("Modulus:       " + (a % b)); // Remainder of division
8 System.out.println("Real division: " + ((double) a / b));

```

Output:

```

Addition:      13
Subtraction:   7
Multiplication: 30
Division:      3
Modulus:       1
Real division: 3.3333333333333335

```

- As in C++, the division operator `/` performs **integer division** when both operands are integers, and the quotient is truncated toward zero: `10 / 3` is 3 and `-7 / 2` is -3 (with remainder `-7 % 2` equal to -1). To obtain a floating-point result, at least one operand must be of type `double`, for example with the **type conversion (cast)** `(double) a / b`.
- The conversion `(int)` removes the decimal part of a `double`: `(int) 3.9` is 3. There is no operator for the power: the method `Math.pow(x, y)` returns x^y as a `double`.

Example 8.8: integer and floating-point division

```

1 int a = 10, b = 3;
2
3 System.out.println(a / b); // Integer division
4 System.out.println(10.0 / 3); // Floating-point division
5 System.out.println((double) a / b); // Conversion of a to double
6 System.out.printf("%.2f\n", (double) a / b); // Two decimals
7 System.out.println(-7 / 2 + " " + -7 % 2); // Truncation toward zero
8 System.out.println((int) 3.9 + " " + Math.pow(2, 10));

```

Output:

```

3
3.3333333333333335
3.3333333333333335
3.33
-3 -1
3 1024.0

```

8.4.2 Assignment operators

Assignment operators, used to assign values to variables:

- `=` (Assignment) $\rightsquigarrow$ Assigns a value to a variable.
- `+=` (Addition assignment) $\rightsquigarrow$ Adds a value and assigns the result.
- `-=` (Subtraction assignment) $\rightsquigarrow$ Subtracts a value and assigns the result.
- `*=` (Multiplication assignment) $\rightsquigarrow$ Multiplies and assigns the result.
- `/=` (Division assignment) $\rightsquigarrow$ Divides and assigns the result.
- `%=` (Modulus assignment) $\rightsquigarrow$ Applies the modulus and assigns the result.

Example 8.9: assignment operators

```
1 int x = 10; // Assignment
2 System.out.println("Initial value: " + x);
3
4 x += 5; System.out.println("After += 5: " + x); // x = x + 5
5 x -= 3; System.out.println("After -= 3: " + x); // x = x - 3
6 x *= 2; System.out.println("After *= 2: " + x); // x = x * 2
7 x /= 4; System.out.println("After /= 4: " + x); // x = x / 4
8 x %= 4; System.out.println("After %= 4: " + x); // x = x % 4
```

Output:

```
Initial value: 10
After += 5: 15
After -= 3: 12
After *= 2: 24
After /= 4: 6
After %= 4: 2
```

Two instructions can be written on the same line, since each of them ends with “;”.

8.4.3 Increment and decrement operators

Increment and decrement operators:

- `++` (Increment) $\rightsquigarrow$ Increases the value of a variable by 1.
- `--` (Decrement) $\rightsquigarrow$ Decreases the value of a variable by 1.
- As in C++, the pre-increment (`++x`) increments the value before returning it, and the post-increment (`x++`) returns the value first, then increments it. The same logic applies to the decrement operator (`--x` and `x--`).

Example 8.10: increment and decrement operators

```

1 int x = 5;
2 System.out.println("Initial value: " + x);
3 System.out.println("Pre-increment (++x): " + ++x);
4 System.out.println("Post-increment (x++): " + x++);
5 System.out.println("Value after post-increment: " + x);
6 System.out.println("Pre-decrement (--x): " + --x);
7 System.out.println("Post-decrement (x--): " + x--);
8 System.out.println("Value after post-decrement: " + x);

```

Output:

```

Initial value: 5
Pre-increment (++x): 6
Post-increment (x++): 6
Value after post-increment: 7
Pre-decrement (--x): 6
Post-decrement (x--): 6
Value after post-decrement: 5

```

8.4.4 Relational operators

Relational operators, used to compare values and return a **boolean** result:

- `==` (Equal to) $\rightsquigarrow$ Checks if two values are equal.
- `!=` (Not equal to) $\rightsquigarrow$ Checks if two values are different.
- `<` (Less than) $\rightsquigarrow$ Checks if one value is smaller than another.
- `>` (Greater than) $\rightsquigarrow$ Checks if one value is larger than another.
- `<=` (Less than or equal to) $\rightsquigarrow$ Checks if one value is smaller than or equal to another.
- `>=` (Greater than or equal to) $\rightsquigarrow$ Checks if one value is larger than or equal to another.
- Relational operators return a value of type **boolean**, displayed as **true** or **false** (not as 1 and 0 as in C++). These operators are commonly used in **conditional statements** and **loops**.

- The operator `==` compares the values of the primitive data types (`int`, `double`, `char`, `boolean`); two `String` values are compared with the method `equals`, for example `name.equals("Alice")`.

Example 8.11: relational operators

```

1 int a = 10, b = 5;
2
3 System.out.println("a == b: " + (a == b)); // Equal to
4 System.out.println("a != b: " + (a != b)); // Not equal to
5 System.out.println("a < b: " + (a < b));    // Less than
6 System.out.println("a > b: " + (a > b));    // Greater than
7 System.out.println("a <= b: " + (a <= b)); // Less than or equal to
8 System.out.println("a >= b: " + (a >= b)); // Greater than or equal to

```

Output:

```

a == b: false
a != b: true
a < b: false
a > b: true
a <= b: false
a >= b: true

```

8.4.5 Logical operators

Logical operators, used for logical operations, often in conditional statements:

- `&&` (Logical AND) $\rightsquigarrow$ Returns `true` if both conditions are `true`.
- `||` (Logical OR) $\rightsquigarrow$ Returns `true` if at least one condition is `true`.
- `!` (Logical NOT) $\rightsquigarrow$ Reverses the Boolean value.

Example 8.12: logical operators

```

1 boolean c1 = true, c2 = false;
2
3 System.out.println("c1: " + c1);
4 System.out.println("c2: " + c2);
5 System.out.println("c1 && c2: " + (c1 && c2)); // True only if both are true
6 System.out.println("c1 || c2: " + (c1 || c2)); // True if at least one is true
7 System.out.println("!c1: " + (!c1));          // Reverses the value
8 System.out.println("!c2: " + (!c2));

```

Output:

```

c1: true
c2: false
c1 && c2: false
c1 || c2: true
!c1: false

```

```
!c2: true
```

8.5 Conditional statements

The relational and logical operators produce the conditions that decide which instructions are executed.

- The `if` statement evaluates a condition inside parentheses (). If the condition is `true`, the block inside the curly braces {} is executed. The optional `else if` clause checks additional conditions, and the optional `else` clause handles all the remaining cases.
- Unlike C++, the condition must be of type `boolean`: an integer is not accepted as a condition, so `if (n % 2)` is a compilation error and the test must be written `if (n % 2 != 0)`.

Example 8.13: conditional statement

```
1 int number = 10;
2
3 // Check if number is positive
4 if (number > 0) {
5     System.out.println("The number is positive.");
6 } else {
7     System.out.println("The number is non-positive.");
8 }
```

Output:

The number is positive.

Example 8.14: if / else if / else

```
1 int score = 85;
2 String grade;
3
4 if (score >= 90) { grade = "A"; }
5 else if (score >= 80) { grade = "B"; }
6 else if (score >= 70) { grade = "C"; }
7 else if (score >= 60) { grade = "D"; }
8 else { grade = "F"; }
9
10 System.out.println("Score: " + score + ", Grade: " + grade);
```

Output:

Score: 85, Grade: B

The conditions are evaluated from top to bottom, and only the block of the first true condition is executed. The variable `grade` is declared without a value: the compiler accepts it because every branch assigns a value to it before it is printed.

8.5.1 Ternary operator (conditional operator)

- The **ternary operator** `?:` is the same as in C++. Syntax: `condition ? value_if_true : value_if_false`.

Example 8.15: ternary operator

```

1  int number = 10;
2
3  // condition ? value_if_true : value_if_false
4  String result = (number > 0) ? "positive" : "non-positive";
5  System.out.println("The number is " + result);
6
7  // Finding the maximum of two numbers
8  int a = 15, b = 20;
9  int maximum = (a > b) ? a : b;
10 System.out.println("Maximum: " + maximum);
11
12 // Checking even or odd
13 int n = 7;
14 String parity = (n % 2 == 0) ? "even" : "odd";
15 System.out.println(n + " is " + parity);
16
17 // Absolute value
18 int x = -5;
19 int absValue = (x >= 0) ? x : -x;
20 System.out.println("Absolute value of " + x + " is " + absValue);

```

Output:

```

The number is positive
Maximum: 20
7 is odd
Absolute value of -5 is 5

```

The class `Math` also provides the methods `Math.max(a, b)` and `Math.abs(x)`, which return the same values.

8.6 Loops

Conditional statements choose *whether* a block of code is executed; loops decide *how many times* it is executed.

- Loops are used to execute a block of code repeatedly while a condition is met. As in C++, the most commonly used loops of Java are the `while` loop and the `for` loop.

8.6.1 The while loop

- The `while` loop executes a block of code as long as the condition inside the parentheses () evaluates to `true`. When the condition is `false`, the loop terminates.

Example 8.16: while loop

```
1 int i = 1; // Initialization
2
3 while (i <= 5) { // Condition
4     System.out.println("Number: " + i);
5     i++; // Increment
6 }
```

Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

8.6.2 The for loop

- The `for` loop combines initialization, condition, and increment (or decrement) in one statement, exactly as in C++.

Example 8.17: for loop

```
1 // Initialization; Condition; Increment
2 for (int i = 1; i <= 5; i++) {
3     System.out.println("Number: " + i);
4 }
```

The output is the same as in the `while` loop of Example 8.16.

8.6.3 The break and continue statements

- The **break** statement immediately terminates the loop.
- The **continue** statement skips the rest of the current iteration and moves to the next one.

Example 8.18: break and continue

```
1 for (int i = 1; i <= 10; i++) {  
2     if (i % 2 == 0) {  
3         continue; // Skip the even numbers  
4     }  
5     if (i > 7) {  
6         break; // Stop the loop at the first odd number greater than 7  
7     }  
8     System.out.print(i + " ");  
9 }  
10 System.out.println();
```

Output:

1 3 5 7

The output is the same as in the C++ and Python versions: the value 9 triggers the **break**.

8.7 Arrays

Loops become really useful when they visit collections of values. A collection of variables of the same data type is stored in an **array**.

- An **array** in Java is an ordered collection of elements of the same data type, with a size fixed at creation time: the size cannot change afterwards.
- The data type of an array of integers is written `int[]`. An array is allocated with the **new** operator, for example `new int[n]`, where the size `n` can be decided during the execution, as for the dynamic-size arrays of C++. The elements of a new array are initialized to 0 (to `false` for `boolean`).
- Elements are accessed using their **index** inside square brackets `[]`, starting from 0: the elements of an array of size `n` are `a[0]`, `a[1]`, ..., `a[n-1]`. Every array knows its size: the length of an array `a` is `a.length`.
- Unlike C++, the memory of an array must not be freed with `delete[]`: it is released automatically by the **garbage collector** of the JVM when the array is no longer used.

Arrays are allocated with **new** before use (or created with a list of values in curly braces). Indices start at 0, the size is fixed and is given by `a.length`.

Example 8.19: creating and accessing an array

```

1  int n = 5;
2
3  int[] a = new int[n]; // Allocation of an array of n integers
4
5  for (int i = 0; i < n; i++) {
6      a[i] = i * 2; // Example of value assignment
7  }
8
9  System.out.println("Length: " + a.length);
10 System.out.println("a[0] = " + a[0]); // First element
11 System.out.println("a[4] = " + a[4]); // Last element, index n-1

```

Output:

```

Length: 5
a[0] = 0
a[4] = 8

```

Accessing an index that does not exist, for example `a[5]`, stops the program with the error `ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5` (in C++ the same access is not checked).

- An array whose values are known when the program is written can be created by listing its values inside curly braces, as in C++.
- The **enhanced for loop** `for (int x : a)` visits the elements of the array `a` directly, without using their indices, like the `for` loop over a list of Python.
- The method `Arrays.toString(a)`, of the class `Arrays` of the package `java.util`, returns the elements of an array as a text in square brackets.

Example 8.20: array literal and iteration

```

1  import java.util.Arrays;
2
3  int[] a = {9, 12, 3, 16, 19, 11, 12};
4  int n = a.length;
5
6  for (int i = 0; i < n; i++) { // Loop over the indices
7      System.out.println("a[" + i + "] = " + a[i]);
8  }
9
10 for (int x : a) { // Loop over the elements
11     System.out.print(x + " ");
12 }
13 System.out.println();
14
15 System.out.println(Arrays.toString(a));

```

Output:

```

a[0] = 9

```

```

a[1] = 12
a[2] = 3
a[3] = 16
a[4] = 19
a[5] = 11
a[6] = 12
9 12 3 16 19 11 12
[9, 12, 3, 16, 19, 11, 12]

```

The `import` instruction is written at the beginning of the file, before the class.

8.7.1 Two-dimensional arrays

- Two-dimensional arrays (matrices) are declared with two pairs of square brackets `[] []`, and allocated with `new int[m][n]` (m rows and n columns) in a single instruction. The element in row i and column j is accessed as `matrix[i][j]`, with both indices starting from 0; `matrix.length` is the number of rows and `matrix[i].length` is the number of columns of row i .
- A two-dimensional array can also be created by listing its values row by row: `int[][] c = {{3, 1, 4}, {1, 5, 9}}`; creates an array with 2 rows and 3 columns, and `c[1][2]` is the element in the second row and third column, equal to 9.

Example 8.21: two-dimensional array (matrix)

```

1 int m = 3; // Number of rows
2 int n = 2; // Number of columns
3
4 int[][] matrix = new int[m][n]; // Allocation of an m x n matrix
5
6 for (int i = 0; i < m; i++) {
7     for (int j = 0; j < n; j++) {
8         matrix[i][j] = i * j; // Example of value assignment
9     }
10 }
11
12 for (int i = 0; i < m; i++) {
13     for (int j = 0; j < n; j++) {
14         System.out.println("matrix[" + i + "][" + j + "] = " + matrix[i][j]);
15     }
16 }

```

Output:

```

matrix[0][0] = 0
matrix[0][1] = 0
matrix[1][0] = 0
matrix[1][1] = 1
matrix[2][0] = 0
matrix[2][1] = 2

```

8.7.2 Lists of variable size: ArrayList

- When the number of elements is not known in advance, the class `ArrayList` of the package `java.util` provides a list that grows during the execution, like the lists of Python. The data type of a list of integers is `ArrayList<Integer>`: the elements of an `ArrayList` are objects, and `Integer` is the class that corresponds to `int`.
- The method `add(x)` appends the element `x` at the end of the list, `get(i)` returns the element of index `i`, `set(i, x)` replaces it, and `size()` returns the number of elements.

Example 8.22: a list of variable size

```

1 import java.util.ArrayList;
2
3 ArrayList<Integer> list = new ArrayList<>(); // Empty list
4
5 for (int i = 1; i <= 4; i++) {
6     list.add(i * i); // The list grows
7 }
8 list.set(0, 100);    // Replaces the element of index 0
9
10 System.out.println(list + " has " + list.size() + " elements");
11 System.out.println("list.get(2) = " + list.get(2));

```

Output:

```

[100, 4, 9, 16] has 4 elements
list.get(2) = 9

```

8.8 Methods (functions)

As in pseudo-code, an algorithm is written as a function that receives the input and returns the output. In Java, functions are called **methods** and must always be defined inside a **class**.

- A method in Java consists of:
 1. an optional **access modifier** (e.g., `public`, `private`) and the keyword `static` when the method belongs to the class rather than to an object of the class (all the methods of this chapter are `static`);
 2. a **return type**: specifies the type of the returned value (`void` if nothing is returned);
 3. a **method name**: the unique identifier of the method;
 4. a **parameter list** (optional): specifies the input values, each one with its data type;
 5. a **body**: the block of code executed when the method is called;
 6. a **return statement**, mandatory if the return type is not `void`.
- The **entry point** of every Java program is the method `public static void main(String[] args)`, which plays the role of the `main` function of C++. The parameter `args` contains the words written after the name of the class in the command `java`.

- Unlike C++, a method can be called before the point where it is defined in the class: the order of the methods inside a class does not matter.
- Java source code can also be compiled and executed online, for example at the following address:

<https://www.mycompiler.io/online-java-compiler>

Example 8.23: methods

```

1 public class Main {
2
3     // Method definition
4     static int addition(int a, int b) {
5         return a + b; // Returns the sum
6     }
7
8     // Method definition
9     static int multiplication(int a, int b) {
10        return a * b; // Returns the product
11    }
12
13    public static void main(String[] args) {
14        int sum = addition(5, 10); // Method call
15        System.out.println("Sum: " + sum);
16
17        int prod = multiplication(5, 10); // Method call
18        System.out.println("Prod: " + prod);
19    }
20 }

```

Output:

Sum: 15
Prod: 50

Example 8.24: method operating on a two-dimensional array

```

1 public class Main {
2
3     static int sumMatrix(int[] [] matrix, int rows, int cols) {
4         int sum = 0;
5         for (int i = 0; i < rows; i++) {
6             for (int j = 0; j < cols; j++) {
7                 sum += matrix[i][j];
8             }
9         }
10        return sum;
11    }
12
13    public static void main(String[] args) {
14        int m = 3, n = 2;
15        int[] [] matrix = new int[m][n];
16    }

```

```

17     for (int i = 0; i < m; i++)
18         for (int j = 0; j < n; j++)
19             matrix[i][j] = i * j;
20
21     int total = sumMatrix(matrix, m, n);
22     System.out.println("Sum of matrix elements: " + total);
23 }
24 }

```

Output:

Sum of matrix elements: 3

The body of a `for` loop made of a single instruction can be written without curly braces, as in C++.

8.8.1 Parameters and return values

- In **Java** every parameter is passed **by value**: the method receives a copy of the argument. For the primitive data types (`int`, `double`, `boolean`, `char`) the changes made to the copy do not affect the variable of the caller; there is no passing by reference as with the symbol `&` of C++.
- A variable of type `int[]` does not contain the elements, but a **reference** to the array (similar to the pointer of C++). The method receives a copy of the reference, which points to the same array: the changes made to the *elements* of the array are visible to the caller.
- A method returns at most one value. When a method has to give back several results, it can return them in an array, for example `new int[]{q, r}`.

Example 8.25: parameters and multiple results

```

1 public class Main {
2
3     static void increment(int x) {
4         x = x + 1; // Modifies only the local copy
5     }
6
7     static void fillWithZeros(int[] a) {
8         for (int i = 0; i < a.length; i++) {
9             a[i] = 0; // Modifies the array of the caller
10        }
11    }
12
13    static int[] divide(int n, int m) {
14        return new int[]{n / m, n % m}; // Two results in an array
15    }
16
17    public static void main(String[] args) {
18        int k = 5;
19        increment(k);
20        System.out.println("After increment: " + k);
21    }

```

```

22     int[] b = {4, 7, 1};
23     fillWithZeros(b);
24     System.out.println("Array: " + b[0] + " " + b[1] + " " + b[2]);
25
26     int[] qr = divide(17, 5);
27     System.out.println("Quotient and remainder: " + qr[0] + " " + qr[1]);
28 }
29 }

```

Output:

After increment: 5
 Array: 0 0 0
 Quotient and remainder: 3 2

8.9 A complete algorithm: the maximum value in an array

We now put all the pieces together: a method, a loop, a conditional statement and an array.

We consider the problem of finding the maximum value in an array a of $n \geq 1$ integer numbers.

- This is the pseudo-code of the algorithm `FindMaximum` of the chapter «Pseudo-code»: the entries are indexed from 1 to n .

Algorithm: FindMaximum

Input : An array $a[1..n]$ of $n \geq 1$ integer numbers

Output: The maximum value in the array a

```

1 function FindMaximum( $a, n$ )
2    $max := a[1]$ 
3   for  $i := 2$  to  $n$  do
4     if  $a[i] > max$  then
5        $max \leftarrow a[i]$ 
6     end
7   end
8   return  $max$ 
9 end

```

- This is the Java implementation. The elements are indexed from 0 to $n - 1$, so the initialization uses `a[0]` and the loop starts from `i = 1`. The size n is not passed as a parameter, because the method obtains it with `a.length`. The method assumes $n \geq 1$, as the pseudo-code does: for an empty array the access to `a[0]` stops the program with an `ArrayIndexOutOfBoundsException`.

```

1 public class Main {
2
3     // Method to find the maximum value in an array
4     static int arrayMaxValue(int[] a) {

```

```

5      int max = a[0]; // Initialize with the first element
6
7      for (int i = 1; i < a.length; i++) {
8          if (a[i] > max) {
9              max = a[i]; // Update the maximum
10         }
11     }
12
13     return max;
14 }
15
16 public static void main(String[] args) {
17     int[] a = {9, 12, 3, 16, 19, 11, 12};
18
19     int max = arrayMaxValue(a);
20     System.out.println("Maximum value: " + max);
21 }
22 }

```

Output:

Maximum value: 19

- The test instance can also be generated at random with the class `Random` of the package `java.util`: if `rand` is an object of this class, `rand.nextInt(100)` returns a random integer between 0 and 99, so `rand.nextInt(100) + 1` is between 1 and 100. The object created with `new Random(2026)` starts from the fixed value 2026 (the **seed**), so that every execution produces the same array; the object created with `new Random()` produces a different array at every execution.

Example 8.26: a random instance

```

1 import java.util.Arrays;
2 import java.util.Random;
3
4 public class Main {
5
6     static int arrayMaxValue(int[] a) {
7         int max = a[0];
8         for (int i = 1; i < a.length; i++) {
9             if (a[i] > max) {
10                 max = a[i];
11             }
12         }
13         return max;
14     }
15
16     public static void main(String[] args) {
17         int n = 10; // Size of the array
18         int[] a = new int[n];
19         Random rand = new Random(2026); // Same random array at every execution
20         for (int i = 0; i < n; i++) {
21             a[i] = rand.nextInt(100) + 1; // Random integer between 1 and 100

```

```

22     }
23
24     System.out.println(Arrays.toString(a));
25     System.out.println("Maximum value: " + arrayMaxValue(a));
26 }
27 }

```

Output:

```

[100, 51, 98, 31, 50, 39, 74, 37, 35, 79]
Maximum value: 100

```

Example 8.27: step-by-step execution

In each table, the column i is the index of the Java loop and the column $i + 1$ is the corresponding index of the pseudo-code. The first row is the initialization $\text{max} = a[0]$.

Instance 1: $a = [9, 12, 3, 16, 19, 11, 12]$, $n = 7$

i (Java)	$i + 1$ (pseudo-code)	$a[i]$	$a[i] > \text{max?}$	max
—	—	—	—	9
1	2	12	Yes	12
2	3	3	No	12
3	4	16	Yes	16
4	5	19	Yes	19
5	6	11	No	19
6	7	12	No	19

Return: $\text{max} = 19$

Instance 2: $a = [3, 7, 2, 9, 1]$, $n = 5$

i (Java)	$i + 1$ (pseudo-code)	$a[i]$	$a[i] > \text{max?}$	max
—	—	—	—	3
1	2	7	Yes	7
2	3	2	No	7
3	4	9	Yes	9
4	5	1	No	9

Return: $\text{max} = 9$

8.10 Bitwise operators

The bitwise operators of the chapter «Introduction to C++» exist in Java with the same symbols and the same meaning: `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (left shift) and `>>` (right shift). The method `Integer.toBinaryString(x)` returns the binary representation of an integer.

- A value of type `int` has 32 bits, so `~12` flips 32 bits and gives `-13`; the 8-bit result of C++ is obtained with the mask `0xFF`, as in Python.
- Java has a third shift operator, `>>>` (unsigned right shift), which always adds zeros on the left, also for negative numbers; for non-negative numbers it coincides with `>>`.
- Since a condition must be of type `boolean`, the test of the least significant bit is written `(n & 1) == 1`, and not `(n & 1)` as in C++.

Example 8.28: bitwise operators

```

1  int a = 12; // Binary: 1100
2  int b = 10; // Binary: 1010
3
4  System.out.println((a & b) + " " + (a | b) + " " + (a ^ b)); // AND, OR, XOR
5  System.out.println(Integer.toBinaryString(a & b));           // 1000
6  System.out.println((5 << 2) + " " + (20 >> 2) + " " + (20 >> 3)); // Shifts
7  System.out.println(~a + " " + (~a & 0xFF)); // NOT, and its 8-bit version
8  System.out.println((-8 >> 1) + " " + (-8 >>> 28)); // Signed and unsigned shift
9
10 int n = 15;
11 System.out.println(((n & 1) == 1) ? "odd" : "even"); // Least significant bit

```

Output:

```

8 14 6
1000
20 5 2
-13 243
-4 15
odd

```

The number `-8` is stored as `11...11000` (32 bits): `-8 >> 1` keeps the sign and gives `-4`, while `-8 >>> 28` keeps only the 4 leftmost bits `1111`, that is `15`.

8.11 Java, C++ and Python side by side

The following table summarizes the main differences among the three languages of this part of the course.

	C++	Java	Python
Execution	compiled (g++)	compiled into bytecode (javac), executed by the JVM (java)	interpreted (python3)
Data types	declared	declared	not declared
Blocks	{ }	{ }	: and indentation
End of instruction	;	;	end of the line
Boolean values	true, false (printed 1, 0)	true, false	True, False
10 / 3	3	3	3.3333333333333335
-7 / 2 (integer)	-3	-3	-7 // 2 is -4
Largest integer	$2^{31} - 1$ for int	$2^{31} - 1$ for int	no limit
Logical operators	&&, , !	&&, , !	and, or, not
Collections	arrays, freed with delete[]	arrays (a.length) and ArrayList, freed automatically	lists (len(a)), freed automatically
Out-of-range index	not checked	exception	exception
Several results	parameters by reference	return of an array	return of a tuple
Entry point	function main	method main of a class	top level of the file

Key points of the chapter

- A Java program is made of classes; it is compiled into bytecode with `javac` and executed by the JVM with `java`, starting from the method `public static void main(String[] args)`.
- Every variable is declared with its data type, which is checked by the compiler; the data type `int` has 32 bits, and the division between two integers is the integer division.
- Conditional statements (`if`, `else if`, `else`, `?:`) and loops (`while`, `for`, and the enhanced `for`) have the same syntax as in C++, but a condition must be of type `boolean`.
- The arrays of Java are allocated with `new`, know their size `a.length`, are indexed from 0 to $n - 1$ and are freed automatically; the class `ArrayList` provides lists of variable size.
- Parameters are passed by value; a method that receives an array can modify its elements, and several results are returned together in an array.