

An Introduction to Computing with Julia

Alfredo Goldman, Lucas de Sousa Rosa

Contents

1. Home	1
2. Acknowledgments	3
3. A Brief History of Computers and Programming Languages	5
3.1 Motivation	5
3.2 Do Computers Need to Exist?	5
3.3 Abacuses, Calculating Machines, and Almost-Computers	5
3.4 The Birth of Modern Computing	6
4. Using the Interpreter (REPL) as a Calculator	9
4.1 Exploring Julia's Interactive Session	9
4.2 Variables and Data Types	12
4.3 Output	14
4.4 External Files and Modules	15
4.5 What is a .jl File?	16
4.6 Check Your Understanding	17
4.7 Explore on Your Own	17
5. Control Structures and Decision Making	19
5.1 Comparison Operators and the Boolean Type	19
5.2 Logical Operators	19
5.3 Changing the Flow of Execution with if-else	20
5.4 The if Structure	20
5.5 Adding Alternatives with else	21
5.6 Multiple Conditions with elseif	21
5.7 Check Your Understanding	21
5.8 Explore on Your Own	21
6. Introduction to Functions	24
6.1 Functions as Natural Abstractions	24
6.2 Functions Calling Other Functions	24
6.3 Accessing Function Documentation	25
6.4 Conversion Functions	25
6.5 Mathematical Functions	25
6.6 Function Overloading	26
6.7 Functions That Return Values	27
6.8 Introduction to Recursion	27
6.9 More Recursion Examples	28
6.10 Check Your Understanding	29
6.11 Explore on Your Own	29
7. More Problems Involving Recursion	32
7.1 Thinking Recursively	32
7.2 The Staircase Problem	32
7.3 Exponentiation by Squaring	33
7.4 Computing the Square Root (Heron's Method)	35
7.5 Binomial Coefficient	36
8. Automated Testing	39
8.1 Why Test?	39
8.2 Writing Tests With Conditionals	39

8.3	Testing With the Test Module	41
8.4	More Examples	41
8.5	Sum of Digits	41
8.6	Checking for Prime Numbers	42
8.7	Check Your Understanding	43
8.8	Explore on Your Own	43
9.	Iterative Repetition Structures	45
9.1	Introduction to while	45
9.2	Comparing Recursion and Iteration	45
9.3	Countdown	45
9.4	Sum of the First N Numbers	46
9.5	Calculating Mathematical Series	47
9.6	When to Use Recursion and Iteration?	48
9.7	Check Your Understanding	48
9.8	Explore on Your Own	48
10.	Exercise Class	50
10.1	Revisiting Factorial: Recursive and Iterative	50
10.2	Approximating the Square Root	50
10.3	Checking Whether a Number Is Prime	51
10.4	Checking Whether a Number Is a Palindrome	52
11.	Revisiting the Previous Class	54
11.1	Randomness	54
12.	Data Input and the Beginning of Lists	59
12.1	The Input Command	59
12.2	Reading Through the Command Line	60
12.3	Lists	61
12.4	Linear Algebra and Lists	62
12.5	Permutation Exercise	63
13.	Vector Exercises	65
13.1	Permutation	65
13.2	Histogram	66
13.3	Modeling Problems with the Computer	67
14.	Modeling Blackjack with Programming	70
14.1	Interactive Version of the Game	70
14.2	Finding the Best Strategy through Simulation	73
15.	Continuing the Modeling	77
16.	Good Practices	83
16.1	Using contracts	83
16.2	Good practice 1: Use types	83
16.3	Automated tests	83
16.4	Good practice 2: Write tests whenever possible	83
16.5	Write code for humans, not for computers	83
16.6	Good practice 3: Write code for others to read	84
16.7	Applying good practices	84
17.	Going Beyond One Dimension (Matrices)	87

18. String Exercises	91
18.1 Concatenating letters	91
18.2 String reversal	91
18.3 The reverse function	92
18.4 Modifying a string	92
18.5 Rearranging letters	93
18.6 Finding the longest word	94
18.7 Returning multiple values	95
19. Appendix: Answers to the question “What is a computer?”	98

1. Home

Welcome to “An Introduction to Computing with Julia”. This book collects the lecture notes for MAC0115 - Introduction to Computing for Exact Sciences and Technology. Written in simple, direct language, it presents the main concepts of programming, complemented by practical code examples.

This course introduces the foundations of computing through a historical journey that leads up to practical programming concepts using the Julia language. It covers algorithms, computer architecture, algorithmic languages (expressions, statements, data structures and functions), development methodologies and good programming practices, along with extensive programming and debugging practice.

The book is being updated alongside the course. The order and the examples may change, but nothing that affects the content taught. Don't worry if something isn't exactly where you expected to find it.

[Download the book as a PDF](#)

2. Acknowledgments

Many people contributed, directly or indirectly, to this book.

The main author is Professor Alfredo Goldman. Text review and technical support were handled by Lucas de Sousa Rosa, his doctoral advisee. We are especially grateful to Beatriz Viana Costa, who helped write some of the chapters. We also thank the students of the MAC0115 course (Introduction to Computing for Exact Sciences and Technology), 2024 class, for their contributions to reviewing the content.

3. A Brief History of Computers and Programming Languages

3.1 Motivation

But, after all, what is a computer? Do we really need to know the answer before we start programming? Actually, no. However, understanding how a computer works makes us better programmers. Defining what a computer is can be more complicated than it seems. We will see that the idea of a “computer” has changed a lot throughout history, always tied to the need to count. Before diving into the history, let’s explore a few different answers to this question.

At the very start of the 2025 classes, in the MAC115 - Introduction to Computing for Exact Sciences and Technology course, we asked the students a simple question: “What is a computer?”. We wrote down the answers (see [Appendix: Answers to the question “What is a computer?”](#)), and despite some funny answers, like “a robot that will control humans one day”, we noticed that most students thought of similar things:

1. A computer is a physical object; a machine or device.
2. It is capable of processing information or data.
3. It’s like a calculator, just more modern.

These ideas capture some of what a computer does, but they don’t fully explain what it is.

3.2 Do Computers Need to Exist?

Professor Douglas Hartree (1897–1958), in his book “Calculating Instruments and Machines”, says that computers can be seen from two angles: anatomical (what they are made of) and physiological (how they work). Defining a computer by its parts would not be enough, since computer parts have changed a great deal over the years.

Around the same time, Professor Arthur L. Samuel proposed a functional definition of what a computer is. According to Samuel, a computer is “a device for performing or facilitating the information process, that is, the handling of information in whatever form it may be presented, according to a well-defined procedure.” This technical view agrees with what the students think. However, this definition leads us to another question: do computers necessarily need to exist as physical devices or machines, or can we conceive of computation beyond the material aspect?

If we asked someone on the street what they think a computer is, they would most likely say it is a “machine” or a “device”. Indeed, this idea comes up often in the answers we collected. However, modern computers were created from an abstract mathematical model called the Turing Machine.

As a child, at age 11, Alan Mathison Turing wrote a letter to his parents, Julius Mathison Turing and Ethel Sara Stoney Turing, describing his idea for how to build a typewriter. Alan Turing did not invent the typewriter, but he used that idea to create the Turing Machine: a mathematical model of computation that manipulates symbols on an infinite tape, following well-defined rules. This model gave rise to modern computers which, according to John A. Robinson, are nothing more than “physical manifestations of one and the same logical abstraction: the universal Turing machine.” So, no. Computers do not need to exist.

3.3 Abacuses, Calculating Machines, and Almost-Computers

This is where we start talking about the history of computing. We humans are very creative. We are always building tools to help us. Archaeological evidence suggests that humans have practiced counting for at least 50,000 years. Counting was used by ancient peoples to track social and economic data, such as the number of people in a group, animals, property, or debts. So it’s no surprise that humans invented tools to help with this process. Around 2700–2300 BC, humans created the abacus: a tool for doing arithmetic faster.

Over the following centuries, other tools were created to help with this. One example is the slide rule, from the 17th century, which did the same thing as the abacus, but using properties of logarithmic functions. The first computers were like more modern versions of these tools, that is, calculating machines. Charles Babbage's Difference Engine (1820), an idealized but never fully built machine, is an example of a calculating machine designed to compute polynomial functions.

The Englishman Charles Babbage was a mathematician, philosopher, inventor, and mechanical engineer, and is considered by many to be the "father of the computer". This title comes from another machine he designed. The Analytical Engine (1840), unlike any machine of its time, marked the transition from mechanized arithmetic to general-purpose computation. This machine would be programmed using punched cards and would incorporate several features later adopted by modern computers, such as sequential control, branching structures, and repetition mechanisms (loops).

While the Analytical Engine was being designed, the Countess of Lovelace, Augusta Ada Byron King (daughter of the poet Lord Byron), created an algorithm to compute the sequence of Bernoulli numbers, and for that reason is considered the first programmer. Today, Ada's figure is used as an inspiration to increase the presence of women in computing, which was much greater a few decades ago.

Up to that point, every machine that had been built was strictly mechanical. The Z2 was one of the first examples of an electrically operated digital computer, built with electromechanical relays, and was created by civil engineer Konrad Zuse in 1940 in Germany. But it was the invention of the vacuum tube (thermionic valve) that gave us the first real leap into the era of digital computers. Vacuum tubes were fully electric devices that generated electric current from the phenomenon of thermionic emission, and were invented by physicist John Ambrose Fleming in 1904.

The Z3, the successor to the Z2, was also developed by Zuse and is considered the first fully automatic, programmable digital computer in the world. Konrad Zuse was also responsible for designing Plankalkül, the first high-level programming language, that is, one closer to human language and farther from machine language. Although this language was never implemented at the time, it introduced fundamental programming concepts such as data types and control structures.

Machines like the Z3, Colossus, and the ENIAC were built by hand, using circuits made of relays or vacuum tubes, and often used punched cards or punched paper tape for input and as their main storage medium. Most machines from this period were not able to store and modify programs (with the exception of the Z3). On the ENIAC, a "program" was defined by the configuration of its connection cables and switches, a feature that distinguishes these machines from modern computers. Programming at that time was carried out mostly by women, though this predominance gradually declined over the following years.

3.4 The Birth of Modern Computing

Up to this point, every machine (Babbage's, Zuse's, Colossus) followed the same design proposed by Babbage: machines were built to perform calculations, and then coded instructions, stored separately in some other form, were organized to make them run. The great paradigm shift happened in 1945 with the ideas of Alan Turing and John von Neumann. Both realized that programs should be stored in the same way as data. The Manchester Baby was the world's first electronic stored-program computer and ran its first program on June 21, 1948. The von Neumann architecture marks the beginning of the era of modern computers. Since then, advances have been made to make them faster, smaller, and easier to use, but their core remains the same: the inherent universality of the stored-program computer.

In the same decade, Kathleen Booth, working in the same place as von Neumann, developed the first assembly language, a programming language very close to machine language. However,

programming in assembly demanded considerable intellectual effort. The languages that came afterward were attempts to abstract machine language into a high-level language, closer to natural, human language. Although several languages were created in the following years, it was only in 1954 that the first widely adopted language appeared: FORTRAN, developed at IBM by a team led by John Backus. Today, more than 70 years later, FORTRAN is still used to rank the TOP500 list of the world's fastest supercomputers.

Starting in 1955, transistor technology revolutionized computing by replacing vacuum tubes in computer design. Transistors had significant advantages: they were smaller and used less power, and consequently generated less heat than their predecessors. The milestone of this transition was the TRADIC Phase One, completed in 1954, considered the first fully transistorized computer.

Technological progress continued with the invention of the integrated circuit in 1958 by Jack Kilby, who was later recognized with the Nobel Prize in the 2000s for this contribution. An integrated circuit consists of a set of electronic circuits made up of several components (transistors, resistors, and capacitors) and their interconnections. These components are carefully etched onto a small, flat piece known as a "chip", made from a semiconductor material, initially germanium and today silicon.

The period from the late 1960s to the late 1970s was marked by the emergence of many programming languages, as well as the consolidation of the main paradigms we know today. Simula, created by Norwegian computer scientists Ole-Johan Dahl and Kristen Nygaard, stood out as the first language specifically designed to support object-oriented programming. At the same time, Dennis Ritchie and Ken Thompson were developing the C language at Bell Labs between 1969 and 1973, aimed at systems programming. The development of C was closely tied to the UNIX operating system, since C was created precisely to make it easier to port UNIX across different hardware platforms. In that same context of innovation, the ML language, conceived by British scientist Robin Milner, emerged as a pioneer among statically typed functional programming languages.

In terms of hardware, the microprocessor enabled the last great leap in the history of computing. Its development was only possible thanks to MOS integrated circuits (CMOS), which allowed the progressive miniaturization of transistors. Today, we are already able to produce transistors on the order of 50 nanometers, which is remarkable when we consider that the radius of a silicon atom is approximately 0.13 nanometers.

On the other hand, the evolution of programming languages was especially notable in the 1990s, driven by the rapid growth of the Internet. During this period, programmer productivity became an important concern, which led to the emergence of several Rapid Application Development (RAD) languages. These languages came together with integrated development environments (IDEs) and garbage collection features. As a result of this trend, languages such as Python, Lua, R, Ruby, Java, JavaScript, and PHP emerged.

The growing popularity of new programming languages was driven by mobile applications. Cell phones and mobile devices became increasingly present in people's daily lives, significantly increasing the demand for apps. Some examples of these languages include Dart, Kotlin, TypeScript, and Swift.

In 2012, the language that will be studied in this course was born: Julia. Aiming to be both fast and productive, Julia combines the performance of older languages with the productivity of more recent ones. It was developed by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. The site <https://julialang.org/benchmarks/> presents a comparative benchmark of different languages solving several algorithms. There, we can see that Julia performs on par with the C language.

4. Using the Interpreter (REPL) as a Calculator

The goal of this chapter is to introduce the Julia interpreter as a powerful calculator and to present the first programming concepts: variables and functions. But first you need to install the Julia language on your computer. More details about the installation process can be found at this [link](#).

Most likely your system is Windows (10 or 11) with a 64-bit architecture. There are a few ways to install Julia on Windows:

1. Through an executable file (.exe).
2. Through terminal commands (winget).

Either option works fine. The first option does not require any additional program, while the second requires a terminal. A terminal is an application that lets you communicate with the operating system through a command-line interface (CLI). The default terminal on Windows is Windows Terminal. It is strongly recommended that you have it installed, which can be done through the Microsoft Store.

Once you have access to a terminal, there are two possible commands to install Julia: `winget install julia -s msstore` or `winget install -e --id JuliaLang.Julia`. Again, either option should work.

4.1 Exploring Julia's Interactive Session

You can open an interactive session (also known as a *read-eval-print loop*, or REPL) of Julia by typing the command `julia` on your terminal's command line. On Windows, after installing the language, you can also open an interactive session by double-clicking the Julia executable. Your window should look like this:

```

      _
     _/
    _/
   _/
  _/
 _/
/

Documentation: https://docs.julialang.org
Type "?" for help, "]"? for Pkg help.
Version 1.11.3 (2025-01-21)
Official https://julialang.org/ release
```

julia>

Inside the session we can enter commands that will be read, evaluated, and printed to the screen. A command is only evaluated when we press **Enter**. Let's start with integer operations. To add two numbers, we can type:

```
1 + 2
3
```

To multiply two other numbers:

```
40 * 4
160
```

As expected, we can use the basic operations of addition (+), subtraction (-), and multiplication (*), and the results are as predicted. However, we'll see next that division behaves in a somewhat particular way:

```
a = 84
b = 2
```

```
# As variáveis a e b são do tipo Int64
```

```
resultado = a / b  
println(resultado)
```

```
42.0
```

Notice that, in this example, a type conversion happened, because 84 and 2 are integers, while the result is a floating-point number (float). Floating-point numbers are binary representations of real numbers, a topic we'll explore in more detail soon. This conversion is evident from the result being displayed as 42.0 instead of simply 42. If you want to obtain the result as an integer, you can use the `div` operator:

```
div(84, 2)
```

```
42
```

Or equivalently using the `\div` operator (to see the division symbol you need to type `\div` followed by the `<tab>` key).

Besides the basic operations, it is also possible to compute exponentiation:

```
2^31
```

```
2147483648
```

More complex expressions can also be evaluated:

```
23 + 2 * 2 + 3 * 4
```

```
39
```

Yes, the usual operator precedence also applies in Julia. However, remember the first lesson of programming: *Write for humans, not for machines*. We can use parentheses to separate the operations:

```
23 + (2 * 2) + (3 * 4)
```

```
39
```

Remember floating-point numbers? All the operations we've seen can also be applied to floats:

```
23.5 * 3.14
```

```
73.79
```

Or:

```
12.5 / 2.0
```

```
6.25
```

The example above shows one more piece of code written clearly for people, since by using `2.0` we make explicit that the second parameter is a floating-point number (float). Floating-point numbers have **limited** precision, so don't be surprised to find unexpected results like the ones shown below:

```
1.2 - 1.0
```

```
0.19999999999999996
```

Errors like this one are quite rare, so much so that we usually place full trust in the calculations performed by computers and calculators. However, there are limitations, as the examples below show.

```
2.6 - 0.7 - 1.9
```

```
2.220446049250313e-16
```

```
0.1 + 0.2
```

```
0.30000000000000004
```

```
10e15 + 1 - 10e15
```

```
0.0
```

These precision issues are related to the limitations of how numbers are represented in a computer. In simplified terms, values in a computer are encoded in words, made up of bits. On modern computers, words have 64 bits, or 8 bytes. This leads to another limitation, related to very large integers.

```
2^63
```

```
-9223372036854775808
```

However, for an introductory course, it is enough to be aware of these limitations. Dealing with these issues in depth is the subject of more advanced courses. The error mentioned above is a *silent error*: when working with integers, the value to be represented may exceed the capacity of the available number of bits, and this failure happens without any explicit notification.

Back to arithmetic. Another interesting operator is `%`, which computes the remainder of a division:

```
4 % 3
```

```
1
```

So far we've seen how to work with a single value, as if we were using the display of a calculator. But we can go beyond that. Instead of simple memory keys, the computer offers us **variables**. These are like names for values that we want to store and use later.

Besides the basic operations, we also have mathematical operations (functions), such as sine. To find out how a function works, we can ask the environment for help, using a `?` or the macro (special functions) `@doc`, followed by what we want to know, as in:

```
@doc sin
```

The output of this command indicates what the function does and even shows some examples:

```
sin(x)
```

Compute sine of x , where x is in radians.

See also `sind`, `sinpi`, `sincos`, `cis`, `asin`.

Examples

```
=====
```

```
julia> round.(sin.(range(0, 2pi, length=9)'), digits=3)
1×9 Matrix{Float64}:
0.0  0.707  1.0  0.707  0.0  -0.707  -1.0  -0.707  -0.0
```

Both `? sin` and `@doc sin` produce the same output.

Notice that not everything shown makes sense right now, but you can already get a sense of how a function like `sin` is used. Now let's look at the square root:

```
@doc sqrt
```

```
sqrt(x)
```

Return $\sqrt{x}$.

Throws `DomainError` for negative Real arguments. Use complex negative arguments instead. Note that `sqrt` has a branch cut along the negative real axis.

The prefix operator $\sqrt{}$ is equivalent to `sqrt`.

See also: `hypot`

...

There we see that we can compute the square root as in:

```
sqrt(4)
```

```
2.0
```

```
sqrt(4.0)
```

```
2.0
```

Now, notice that the documentation for the `big()` function reads as follows:

```
big(T::Type)
```

Compute the type that represents the numeric type `T` with arbitrary precision. Equivalent to `typeof(big(zero(T)))`.

Examples

=====

```
julia> big(Rational)
Rational{BigInt}
```

```
julia> big(Float64)
BigFloat
```

```
julia> big(Complex{Int})
Complex{BigInt}
```

```
big(x)
```

Convert a number to a maximum precision representation (typically `BigInt` or `BigFloat`). See `BigFloat` for information about some pitfalls with floating-point numbers.

The `big()` function lets us create numbers of large magnitude, represented by the `BigInt` or `BigFloat` types. This function is particularly useful when you need to work with numbers that are much larger than the limits of standard types like `Int64` or `Int32`. By using `BigInt` numbers, we eliminate overflow problems, as we can see below:

```
numero_grande = big(2) ^ 1002
(digitos = ndigits(numero_grande), excede_int64 = numero_grande > typemax(Int64))
(digitos = 302, excede_int64 = true)
```

4.2 Variables and Data Types

As already introduced, Julia has the concept of variables. Variables are used to store various kinds of data, such as integers and floats. We can operate on variables the same way we operate on the data they hold (see the example below).

```
a = 7
2 + a
9
```

When we write `a = 7`, we are performing an operation called **assignment**. The `=` operator in Julia (and in most programming languages) does not represent mathematical equality, but rather an instruction to store the value on the right into the variable on the left. We can picture this as if we were putting the value 7 inside a box called `a`.

Variables in Julia can be assigned new values, and the type of the variable is determined by the last assignment made. The `typeof` function can be used to identify the type of a given variable.

```
a = 3
typeof(a)
Int64

a = a + 1
typeof(a)
Int64
```

Assignment always happens from right to left: first the expression on the right is evaluated, and then that value is stored in the variable on the left. In the example below, the variable `b` starts out with an integer value. However, after the multiplication operation, its value becomes a floating-point type.

```
b = 3
b = b * 0.5
typeof(b)
Float64
```

The ability to change a variable's type is known as **dynamic typing**. This feature offers several advantages, such as the flexibility to reuse variables to store different types of data over time and reduced verbosity, since it isn't necessary to specify the type of each variable, which improves code readability. In this context, we can observe that Julia has several primitive types, the main ones being:

```
typeof(1)
Int64

typeof(1.1)
Float64

typeof("Bom dia")
String
```

Speaking of **strings**, they are defined as sets of characters between quotation marks, like:

```
s1 = "Olha que legal"
s2 = "Outra String"

"Outra String"
```

It is also possible to perform operations on strings, such as **concatenation**:

```
s1 = "Tenha um"
s2 = " Bom dia"
s3 = s1 * s2

"Tenha um Bom dia"
```

Or repetition using the power operator:

```
s = "Não vou mais fazer coisas que possam desagradar os meus colegas "  
s ^ 10
```

```
"Não vou mais fazer coisas que possam desagradar os meus colegas Não vou mais  
fazer coisas que possam desagradar os meus colegas Não vou mais fazer coisas que  
possam desagradar os meus colegas Não vou mais fazer coisas que possam desagradar  
os meus colegas Não vou mais fa" ... 100 bytes ... " meus colegas Não vou mais fazer  
coisas que possam desagradar os meus colegas Não vou mais fazer coisas que possam  
desagradar os meus colegas Não vou mais fazer coisas que possam desagradar os meus  
colegas Não vou mais fazer coisas que possam desagradar os meus colegas "
```

To avoid typing too many characters, we can sometimes use *syntactic sugar*.

```
x = 1  
x = x + 1  
x += 1 # '+=' equivale a '= x + 1', também funciona para os operadores *, - e /  
3
```

The code above uses comments (everything after #). These comments are ignored by the interpreter and can be used to make the code more readable.

Still on the subject of variables, there are some rules regarding their names: they must start with a letter (or with `_`), can contain digits, and cannot be reserved words. Julia, being a modern language, also accepts Unicode characters and emojis in names, such as Δ (`\Delta`).

```
 $\Delta$  = 2  
2  
 = 5 # \:cat: <tab>  
 = 3 # \:dog: <tab>  
 = 20 # \:house: <tab>  
20
```

This doesn't add anything from an algorithmic standpoint, but it does let you have some rather charming variables. The list of figures can be found [here](#).

4.3 Output

To print information to the terminal, we use the `print()` and `println()` functions. The difference between them is that the first does not move to a new line, while the second does.

```
print("Hello ")  
println("World!")  
println("Ola, mundo!")  
Hello World!
```

```
Ola, mundo!
```

The `println()` command can take multiple arguments, which will be automatically converted to strings and concatenated:

```
nome = "Maria"  
idade = 25  
println("Olá, meu nome é ", nome, " e tenho ", idade, " anos.")  
Olá, meu nome é Maria e tenho 25 anos.
```

For more complex formatting, Julia offers **string interpolation**, where we can insert variables and expressions directly inside a string using the dollar sign \$:

```
nome = "João"
altura = 1.75
println("$nome tem $altura metros de altura.")
João tem 1.75 metros de altura.
```

We can also include expressions inside braces after the dollar sign:

```
preco = 9.99
quantidade = 3
println("Total da compra: R\$ $(preco * quantidade)")
Total da compra: R$ 29.97
```

For numeric formatting, we can use the `@sprintf` function or the `@printf` macro from the `Printf` module (more details about modules in [External Files and Modules](#)):

```
using Printf

valor = 123.456
@printf("Valor formatado: %.2f\n", valor) # Exibe com 2 casas decimais
Valor formatado: 123.46
```

Or alternatively:

```
valor = 123.456
s = @sprintf("Valor formatado: %.2f", valor)
println(s)
Valor formatado: 123.46
```

4.4 External Files and Modules

In the previous example we used the syntax `using Printf`. This is the syntax for importing a module in Julia. Modules are organized collections of code that we can use in our programs. The `Printf` module is part of Julia's **standard library** and offers functions for formatting types in the style of the C language. By writing `using Printf`, we tell the interpreter that we want to access the functions of this module, such as `@printf` and `@sprintf`. To find out which functions are available in this and other modules, check Julia's official documentation. The specific documentation for the `Printf` module is available at <https://docs.julialang.org/en/v1/stdlib/Printf/>.

Julia ships with several standard modules that can be useful:

- **Statistics**: for statistical calculations (mean, median, etc.)
- **Dates**: for working with dates and times
- **Printf**: for advanced text formatting
- **LinearAlgebra**: for linear algebra operations
- **Random**: for random number generation

The Julia community also develops many modules (packages) that can be installed to expand the language's functionality. We'll learn how to do that later in the course.

Besides modules, Julia lets you load code from external files using the `include()` command. This command reads the specified file and executes its entire contents in the current context. As a result, all the functions, variables, and definitions from the file become available in the environment where `include` was called.

On Windows, file paths traditionally use backslashes (\). In Julia, however, we can use either forward slashes (/) or backslashes. There is one important detail: when we use backslashes inside strings in Julia, we need to double them. This happens because a lone backslash (\) is a special character in strings, used to represent characters like \n (newline) or \t (tab). To indicate that we want a literal backslash, we need to write two of them (\\). Because of this, paths with several folders become harder to read:

```
# Caminho usando barras normais (recomendado)
include("C:/Users/MeuUsuario/Documentos/arquivo.jl")

# Mesmo caminho usando barras invertidas (mais complicado)
include("C:\\Users\\MeuUsuario\\Documentos\\arquivo.jl")
```

If the file is in the same directory as your current script or REPL, you can simply use the file name:

```
include("funcoes.jl")
```

For files in subdirectories of the current directory:

```
include("utilitarios/matematica.jl")
```

For files in the parent directory:

```
include("../exemplos.jl")
```

Let's look at a practical example. Suppose you created a file called `funcoes.jl` in the folder `C:/Projetos/Julia/` with the following content:

```
function ola(nome)
    println("Olá ", nome)
end

function soma(a, b)
    return a + b
end
```

Now you can use these functions (more on functions in [Introduction to Functions](#)) in the REPL or in another file:

```
# No REPL ou em um arquivo na mesma pasta:
include("funcoes.jl")

# Ou com caminho completo:
include("C:/Projetos/Julia/funcoes.jl")

# Agora podemos usar as funções definidas em funcoes.jl
ola("Alfredo")           # Imprime: Olá Alfredo
resultado = soma(5, 3)    # resultado = 8
println(resultado)        # Imprime: 8
```

This feature is especially useful for organizing your code across multiple files, letting you split larger programs into smaller pieces.

4.5 What is a .jl File?

A `.jl` file is similar to a `.txt` text file, but with the `.jl` extension. Although it's possible to open it with a simple text editor like Notepad, doing so is not recommended for programming. `.jl` files are Julia source code files and are generally edited with dedicated programming editors, such as Visual Studio Code, Atom, or Sublime Text.

There isn't one text editor that's definitively better than the others, what matters is choosing the one you feel most comfortable with. Our recommendation is Visual Studio Code, which offers much more advanced features than a plain text editor and has an extension dedicated to the Julia language, making it significantly easier to write code. To get started using Visual Studio Code with Julia, the following tutorials may be helpful:

- <https://code.visualstudio.com/docs/getstarted/getting-started>
- <https://code.visualstudio.com/docs/languages/julia>

4.6 Check Your Understanding

1. What is the difference between the results obtained from the `/` and `div` operators in Julia? In which situations would each be more appropriate?
2. Why doesn't the expression `2.6 - 0.7 - 1.9` result in exactly zero? What does this teach us about computational calculations?
3. Explain what "dynamic typing" means and how it affects the behavior of variables when we assign different types of values to them.
4. Use the `big()` function to compute 2^{1000} . Compare this result with what happens when you try to compute 2^{1000} without using `big()`.
5. Store your first and last name in separate variables, then combine them to form your full name with a space between them. Also demonstrate string repetition.
6. Create the variables `a = 10`, `b = 3`, and `c = 4.5`. Perform the following calculations: `a + b + c`, `a * b * c`, `a % b`, and check the type of the result of each operation using `typeof()`.

4.7 Explore on Your Own

1. Look through the documentation for two mathematical functions that were not mentioned in this chapter and test their use in the REPL.
2. What happens when you try to divide a number by zero in Julia? And when you compute `0/0`? Try it and observe the results.
3. Try the Unicode operator $\approx$ (type `\approx` followed by **TAB**). How does it behave when comparing `0.1 + 0.2 ≈ 0.3`?
4. Investigate the `round()` function and use it to fix some of the precision issues demonstrated in this chapter.

5. Control Structures and Decision Making

The goal of this chapter is to understand how a program can make decisions and change its flow of execution. We will explore comparison operators, the Boolean type, and the `if`, `else`, and `elseif` conditional structures in Julia.

5.1 Comparison Operators and the Boolean Type

Before studying conditional structures, we need to understand comparison operators and the data type they produce: the Boolean type (`Bool`). A Boolean variable can only have two possible values: `true` or `false`. Let's look at the main comparison operators in Julia:

```
# Igualdade: retorna true se os valores forem iguais
2 + 2 == 4

true

# Diferença: retorna true se os valores forem diferentes
3 != 8

true

# Menor que: retorna true se o primeiro valor for menor que o segundo
23 < 24

true

# Menor ou igual: retorna true se o primeiro valor for menor ou igual ao segundo
42 <= 44

true

# Maior que: retorna true se o primeiro valor for maior que o segundo
42 > 2

true

# Maior ou igual: retorna true se o primeiro valor for maior ou igual ao segundo
42 >= 42

true
```

In programming languages, including Julia, the equal sign (`=`) is used to assign values to variables, while the equality operator (`==`) is used for comparisons.

We can check the type of a comparison expression:

```
typeof(2 == 3)

Bool
```

As expected, the type is `Bool`, indicating a Boolean value.

5.2 Logical Operators

In addition to comparison operators, Julia also provides logical operators that let us combine or modify Boolean values:

```
# Operador NOT (negação): inverte o valor booleano
!true

false

!false

true

# Operador AND: retorna true apenas se ambos os valores forem true
true && true
```

```
true
true && false
false
# Operador OR: retorna true se pelo menos um dos valores for true
true || false
true
false || false
false
```

These operators are essential for building more complex conditions in our conditional structures.

5.3 Changing the Flow of Execution with if-else

So far, our programs have followed a linear flow of execution, with instructions being executed in the order they were written. Consider the example:

```
println("Oi")
println("um")
println("dois")

Oi
um
dois
```

The order of printing will be “Oi”, “um”, and “dois”, exactly in the sequence in which the commands were written.

However, we often need our program to make decisions and execute different blocks of code depending on certain conditions. This is where the `if` conditional structure comes in.

5.4 The if Structure

The `if` structure lets us execute a block of code only if a condition is true:

```
pandemia = true
println("Vou sair de casa?")
if pandemia == true
  println("Só vou sair de casa se for essencial")
end
```

```
Vou sair de casa?
Só vou sair de casa se for essencial
```

In this example, the message “Só vou sair de casa se for essencial” will only be printed if the variable `pandemia` is equal to `true`.

Here is another example:

```
denominador = 1
if denominador != 0
  println("Sei fazer a divisão se não for por zero")
  println("0 resultado da divisão de 30 por ", denominador, " é igual a ", 30/
denominador)
end
```

```
Sei fazer a divisão se não for por zero
0 resultado da divisão de 30 por 1 é igual a 30.0
```

The code inside the `if` block will only run if the denominator is different from zero, thus avoiding a division-by-zero error.

5.5 Adding Alternatives with else

We often want to execute one block of code if a condition is true and another block if the condition is false. For this, we use the **if-else** structure:

```
pandemia = true
println("Vou sair de casa?")
if pandemia == true
    println("Só vou sair de casa se for essencial")
else
    println("Balada liberada!!")
end
```

```
Vou sair de casa?
Só vou sair de casa se for essencial
```

If the variable `pandemia` is true, the message “Só vou sair de casa se for essencial” will be printed. Otherwise, the message “Balada liberada!!” will be printed.

5.6 Multiple Conditions with elseif

What if we have more than two possible situations? In that case, we can use the **if-elseif-else** structure:

```
pandemia = true
tenhoqueestudar = true
println("Vou sair de casa?")
if pandemia == true
    println("Só vou sair de casa se for essencial")
elseif tenhoqueestudar == true
    println("Melhor ficar em casa")
else
    println("Balada liberada")
end
```

```
Vou sair de casa?
Só vou sair de casa se for essencial
```

In this example, we have three possible paths:

1. If there is a pandemic, only go out if it is essential
2. If there is no pandemic but I have to study, stay home
3. If there is no pandemic and I don't have to study, go to the party

The **if-elseif-else** structure evaluates conditions in the order they appear. As soon as a true condition is found, the corresponding block is executed and the remaining conditions are ignored.

5.7 Check Your Understanding

1. What is the difference between the `=` operator and the `==` operator in Julia? Why does this distinction matter?
2. Explain the difference between **if-else** and **if-elseif-else**. In which situations would you use each one?
3. Consider the following Boolean expression: `(a > b) && !(c == d)`. Explain in words what it means.

5.8 Explore on Your Own

1. Research short-circuit evaluation of the `&&` and `||` logical operators in Julia. How can this behavior be useful in programming?
2. In Julia, besides the values `true` and `false`, what other values are considered “truthy” or “falsy” in a Boolean context?

3. Investigate the ternary operator (`?:`) in Julia and how it can be used as a more concise alternative to certain **if-else** structures.
4. Explore how conditional structures can be combined with functions to create more modular and reusable code.

6. Introduction to Functions

The goal of this chapter is to understand the concept of functions in programming and how they are implemented in Julia. We will explore how to create our own functions, how they can receive parameters and return values, and we will also introduce the concept of recursion.

6.1 Functions as Natural Abstractions

In the previous class, we already used some predefined functions in Julia. Functions are blocks of code that perform specific tasks and can be reused whenever needed. They let us abstract complex operations into simple commands, making code more readable and modular.

Let's recall some of the functions we've already used:

- `typeof()` - Takes a value as a parameter and returns its type.
- `div()` - Takes two numbers and returns the integer division of the first by the second.
- `print()` and `println()` - Print values to the console, with the second one adding a line break after printing.

So far, we've seen examples of **function calls** like `sin(0.5)` or `sqrt(4)`. When we write the name of a function followed by parentheses containing arguments, we are "calling" or "invoking" that function.

But how are these functions created? In Julia, we can declare our own functions using the `function` keyword:

```
function dobro(x)
    return x * 2
end
```

`dobro` (generic function with 1 method)

Here, `dobro` is the name of the function, `x` is a parameter (a value that the function receives), and `return x * 2` specifies what the function should compute and return when called. Providing parameters to a function is optional.

After declaring the function, we can call it multiple times with different arguments:

```
resultado1 = dobro(5)      # Chama a função com o argumento 5
println(resultado1)        # Imprime 10

resultado2 = dobro(3.5)    # Chama a função com o argumento 3.5
println(resultado2)        # Imprime 7.0

10
7.0
```

Declaring a function and calling it are two different actions:

- **Declaration:** Defines how the function should behave (happens once)
- **Call:** Executes the function with specific values (can happen multiple times)

Functions like `sin()`, `sqrt()`, and `big()` already come declared in Julia, which is why we can use them directly.

6.2 Functions Calling Other Functions

A function can call another function, allowing for the composition of more complex operations:

```
function imprime(a)
    println("Vou imprimir ", a)
end
```

```
function imprimeduasvezes(a)
    imprime(a)
    imprime(a)
end
```

imprimeduasvezes (generic function with 1 method)

Let's test our new function:

```
imprimeduasvezes(13)
```

Vou imprimir 13

Vou imprimir 13

6.3 Accessing Function Documentation

We can ask the interpreter for help to better understand how these functions work. To do this, we use the question mark `?` or the `@doc` macro before the function name:

```
# Exemplos de como acessar a documentação
@doc typeof
@doc div
@doc println
```

By checking the documentation, we discover that some functions such as `div()` can be used with an alternative syntax, such as `\div`. This kind of notation is particularly useful for mathematical operations.

6.4 Conversion Functions

An important category of functions in Julia are conversion functions, which transform values from one type into another. Let's look at some examples:

```
# Converte uma string para um número em ponto flutuante
parse(Float64, "32")
```

32.0

```
# Converte um número em ponto flutuante para um inteiro (removendo a parte decimal)
trunc{Int64}(2.25)
```

2

```
# Converte um inteiro para um número em ponto flutuante
float(2)
```

2.0

```
# Converte um número para uma string
string(3)
```

"3"

```
# Converte um número em ponto flutuante para uma string
string(3.57)
```

"3.57"

6.5 Mathematical Functions

Julia has a large library of ready-to-use mathematical functions. Here are some of the most common ones:

Function	Description
<code>sin(x)</code>	Computes the sine of (x) in radians
<code>cos(x)</code>	Computes the cosine of (x) in radians
<code>tan(x)</code>	Computes the tangent of (x) in radians
<code>deg2rad(x)</code>	Converts (x) from degrees to radians
<code>rad2deg(x)</code>	Converts (x) from radians to degrees
<code>log(x)</code>	Computes the natural logarithm of (x)
<code>log(b, x)</code>	Computes the logarithm of (x) in base (b)
<code>log2(x)</code>	Computes the base-2 logarithm of (x)
<code>log10(x)</code>	Computes the base-10 logarithm of (x)
<code>exp(x)</code>	Computes the natural exponential of (x)
<code>abs(x)</code>	Computes the absolute value of (x)
<code>sqrt(x)</code>	Computes the square root of (x)
<code>cbrt(x)</code>	Computes the cube root of (x)
<code>factorial(x)</code>	Computes the factorial of (x)

A good way to get familiar with these functions is to experiment with different values and check the results. For more complex functions, there may already be ready-made implementations in Julia. A useful tip is to search the internet using keywords like “julia lang hyperbolic sin” to find the function you need. In general, searching in English tends to produce better results.

6.6 Function Overloading

In Julia, we can have functions with the same name but with different numbers or types of parameters. This is called “function overloading”:

```
function recebe(a)
    println("Recebi um parâmetro: ", a)
end

function recebe(a, b)
    println("Recebi dois parâmetros: ", a, " e ", b)
end
```

recebe (generic function with 2 methods)

The interpreter decides which version of the function to call based on the arguments provided:

```
recebe(1)
Recebi um parâmetro: 1

recebe(1, 2)
Recebi dois parâmetros: 1 e 2
```

We can also call functions using variables and expressions as arguments:

```
a = 10
recebe(a)
recebe(a, a + 1)

Recebi um parâmetro: 10
Recebi dois parâmetros: 10 e 11
```

6.7 Functions That Return Values

So far, we've seen functions that only print messages but don't return any value. The return type of these functions is `Nothing`, indicating that they don't produce a value that can be assigned to a variable.

However, we often want our functions to compute and return values. For this, we use the `return` keyword:

```
function somal(a)
    return a + 1
end
```

`somal` (generic function with 1 method)

Now we can use this function in expressions and assignments:

```
resultado = somal(5)
println("O resultado é: ", resultado)
```

O resultado é: 6

```
# Também podemos usar o resultado em outras expressões
println("Resultado multiplicado por 2: ", somal(5) * 2)
```

Resultado multiplicado por 2: 12

We can create functions for more complex calculations:

```
function hipotenusa(a, b)
    hip = sqrt(a^2 + b^2)
    return hip
end
```

`hipotenusa` (generic function with 1 method)

Let's test our function:

```
# Calculando a hipotenusa de um triângulo 3-4-5
hipotenusa(3, 4)
```

5.0

6.8 Introduction to Recursion

Now let's explore a fundamental concept in programming: recursion. A recursive function is one that calls itself as part of its execution. This may seem strange at first, but it's a powerful technique for solving certain kinds of problems.

Let's start with a simple example: computing the factorial of a number. The factorial of n (written $n!$) is the product of all positive integers less than or equal to n . For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

The factorial can be defined recursively as:

- Base case: $0! = 1$
- Recursive case: $n! = n \times (n - 1)!$

Let's implement this in Julia:

```
function fatorial(n)
    if n == 0
        return 1 # Caso base
    else
        return n * fatorial(n - 1) # Chamada recursiva
    end
end
```

```
end
end

fatorial (generic function with 1 method)
```

Let's test our function:

```
fatorial(5)

120
```

To understand how recursion works, let's trace through the computation of `fatorial(3)` step by step:

1. We call `fatorial(3)`
 - Since 3 is not equal to 0, we execute `return 3 * fatorial(2)`
2. Now we need to compute `fatorial(2)`
 - Since 2 is not equal to 0, we execute `return 2 * fatorial(1)`
3. Now we need to compute `fatorial(1)`
 - Since 1 is not equal to 0, we execute `return 1 * fatorial(0)`
4. Now we need to compute `fatorial(0)`
 - Since 0 equals 0, we return 1
5. Now we can complete the computation of `fatorial(1) = 1 × 1 = 1`
6. Now we can complete the computation of `fatorial(2) = 2 × 1 = 2`
7. Finally, we complete the computation of `fatorial(3) = 3 × 2 = 6`

Recursion has two fundamental parts:

1. A **base case** that ends the recursion (in our example, when $n = 0$)
2. A **recursive case** that moves the problem closer to the base case (in our example, by reducing n by 1)

The recursion must always reach the base case, otherwise the function will keep calling itself indefinitely, causing a stack overflow error.

6.9 More Recursion Examples

Let's implement a recursive function for a countdown:

```
function contagem(n)
    if n < 0
        println("Fim!")
    else
        print(n, " ")
        contagem(n - 1)
    end
end

contagem (generic function with 1 method)
```

Let's test our function:

```
contagem(5)

5 4 3 2 1 0 Fim!
```

We can also use recursion to compute the sum of the first n integers:

```
function soma(n)
  if n == 0
    return 0 # Caso base
  else
    return n + soma(n - 1) # Caso recursivo
  end
end
```

soma (generic function with 1 method)

Let's test our function:

```
soma(10)
```

55

Another interesting example is computing the sum of the terms of the harmonic series:

```
function somaharmonica(atual, n)
  # Caso base: quando chegamos ao último termo
  if atual > n
    return 0.0
  else
    # Caso recursivo: somamos o termo atual e chamamos a função para o próximo
    termo
    return 1.0 / atual + somaharmonica(atual + 1, n)
  end
end
```

somaharmonica (generic function with 1 method)

Let's compute the sum of the first 10 terms of the harmonic series:

```
somaharmonica(1, 10)
```

2.9289682539682538

6.10 Check Your Understanding

1. What is the difference between a function that prints a value and a function that returns a value? Why does this distinction matter?
2. Explain the concept of recursion in your own words. What are the essential components of a recursive function?
3. Write a function that takes a positive integer and returns the sum of its digits. For example, for the number 123, the function should return $1 + 2 + 3 = 6$.
4. Implement a function that computes the n -th number of the **Fibonacci sequence** using recursion. Recall that
 - $Fib(0) = 0$
 - $Fib(1) = 1$
 - $Fib(n) = Fib(n - 1) + Fib(n - 2)$, if $n > 1$.
5. Write a function that takes two numbers as parameters and returns their **greatest common divisor (GCD)** using Euclid's algorithm recursively.

6.11 Explore on Your Own

1. Research the concept of the "call stack" and how it relates to recursion. What are the practical limitations of recursion due to the call stack?
2. Think about how you could optimize the recursive Fibonacci function to avoid repeated computations.
 - Hint: look up "memoization".

3. Explore functions with a variable number of arguments in Julia using “splat” syntax (...).
4. Find out how you can define default values for function parameters in Julia.

7. More Problems Involving Recursion

In the previous chapter, we introduced the concept of recursion and saw how functions can call themselves to solve problems. In this chapter, we'll explore a few more problems that can be solved using recursion, which will help us develop our ability to "think recursively."

7.1 Thinking Recursively

Solving problems using recursion requires a shift in how we look at problems. Instead of thinking about all the steps needed to reach the solution, we focus on:

1. How to express the problem in terms of a smaller version of the same problem
2. When to stop the recursion (base case)

Let's explore four problems that illustrate the power of recursion well: the staircase problem, exponentiation by squaring, computing a square root using Heron's method, and the binomial coefficient.

7.2 The Staircase Problem

Imagine a staircase with n steps. You can climb 1 or 2 steps at a time. In how many different ways can you reach the top of the staircase?

For example, if we have a staircase with 3 steps, there are 3 ways to climb it:

- Take three steps of 1: (1, 1, 1)
- Take a step of 1 followed by a step of 2: (1, 2)
- Take a step of 2 followed by a step of 1: (2, 1)

How can we think about this problem recursively? To reach step n , we must have come from step $n - 1$ (by taking a step of 1) or from step $n - 2$ (by taking a step of 2). Therefore, the total number of ways to reach step n is the sum of the number of ways to reach step $n - 1$ and step $n - 2$.

Our base cases are:

- If $n = 0$ (no steps): there is 1 way (don't climb at all)
- If $n = 1$ (one step): there is 1 way (take a step of 1)

Let's implement this solution:

```
function maneiras_subir_escada(n)
  # Casos base
  if n == 0 || n == 1
    return 1
  else
    # Caso recursivo: soma das maneiras de chegar a partir de n-1 e n-2
    return maneiras_subir_escada(n - 1) + maneiras_subir_escada(n - 2)
  end
end
```

maneiras_subir_escada (generic function with 1 method)

Let's test our code for different numbers of steps:

```
for i in 1:10
  println("Escada com $i degraus: $(maneiras_subir_escada(i)) maneiras diferentes")
end
```

Escada com 1 degraus: 1 maneiras diferentes

Escada com 2 degraus: 2 maneiras diferentes
Escada com 3 degraus: 3 maneiras diferentes
Escada com 4 degraus: 5 maneiras diferentes
Escada com 5 degraus: 8 maneiras diferentes
Escada com 6 degraus: 13 maneiras diferentes
Escada com 7 degraus: 21 maneiras diferentes
Escada com 8 degraus: 34 maneiras diferentes
Escada com 9 degraus: 55 maneiras diferentes
Escada com 10 degraus: 89 maneiras diferentes

Note that we're using a different structure here, for. Don't worry about the details of this structure for now, but know that we're simply repeating the execution of a block of code.

If you look closely at this sequence of results, you'll notice that it corresponds to the famous Fibonacci sequence: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...). This is no coincidence. The staircase problem and the Fibonacci sequence share the same recursive structure.

7.3 Exponentiation by Squaring

When we want to compute powers like a^n , the simplest approach would be to multiply a by itself n times. For example, to compute 2^8 , we would do $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$, performing 7 multiplications. But there is a much more efficient approach using recursion.

The exponentiation-by-squaring method we're about to present can compute the same value using only about $\log_2 n$ operations. For example, to compute 2^8 , we would need only 3 multiplications. For larger numbers, this difference is even more significant: computing 2^{1000} would require 999 multiplications with the simple method, but only about 10 multiplications with our recursive method.

The idea is based on the following mathematical properties:

- If n is negative: $a^n = 1/a^{-n}$ (we invert the result of the positive power)
- If n is even: $a^n = (a^{n/2})^2$ (we compute "half" the power and square it)
- If n is odd: $a^n = a \times a^{n-1}$ (we multiply by an even power, which we know how to compute)

How does this translate into recursive thinking?

1. If we want to compute a^n and n is even:
 - First we compute $a^{n/2}$ (a smaller problem)
 - Then we multiply that result by itself
2. If we want to compute a^n and n is odd:
 - First we compute a^{n-1} (which is even, and we know how to solve it using the previous case)
 - Then we multiply that result by a
3. If we want to compute a^n and n is negative:
 - We compute a^{-n} (we know how to solve this using the previous cases)
 - Then we compute $1/a^{-n}$

Our base cases (where the recursion stops) are:

- If $n = 0$, then $a^n = 1$ (any number raised to 0 is 1)
- If $n = 1$, then $a^n = a$ (any number raised to 1 is itself)

Let's implement this solution:

```
function potenciacao(base, expoente)
  # Resolvemos o expoente negativo primeiro
```

```

    if expoente < 0
        return 1 ÷ potenciacao(base, -expoente)
    end

    if expoente == 0
        return 1
    elseif expoente == 1
        return base
    end

    # Se o expoente for par
    if expoente % 2 == 0
        temp = potenciacao(base, expoente ÷ 2)
        return temp * temp
    else
        # Se o expoente for ímpar
        return base * potenciacao(base, expoente - 1)
    end
end

```

potenciacao (generic function with 1 method)

Let's test our function:

```

println(potenciacao(2, 10)) # Deve retornar 1024
println(potenciacao(3, 5))  # Deve retornar 243
1024
243

```

To better understand how this approach saves operations, let's trace the execution of `potenciacao(2, 10)`:

Step 1: `potenciacao(2, 10)`

- `expoente = 10` is even
- We need to compute `potenciacao(2, 5)^2`

Step 2: `potenciacao(2, 5)`

- `expoente = 5` is odd
- We need to compute `2 * potenciacao(2, 4)`

Step 3: `potenciacao(2, 4)`

- `expoente = 4` is even
- We need to compute `potenciacao(2, 2)^2`

Step 4: `potenciacao(2, 2)`

- `expoente = 2` is even
- We need to compute `potenciacao(2, 1)^2`

Step 5: `potenciacao(2, 1)`

- `expoente = 1` is odd
- We need to compute `2 * potenciacao(2, 0)`

Step 6: `potenciacao(2, 0)`

- base case, returns 1

Returned values:

- `potenciacao(2, 0)` returns 1
- `potenciacao(2, 1)` returns $2 * 1 = 2$
- `potenciacao(2, 2)` returns $2^2 = 4$
- `potenciacao(2, 4)` returns $4^2 = 16$
- `potenciacao(2, 5)` returns $2 * 16 = 32$
- `potenciacao(2, 10)` returns $32^2 = 1024$

For computing enthusiasts, when we talk about **complexity** we're referring to the efficiency of an algorithm in terms of the number of operations performed. To represent the amount of operations, we use *Big-O* notation, symbolized as $O(\cdot)$.

In this context, the approach we used manages to reduce the complexity from $O(n)$ (where the number of operations grows **linearly** with the size of the input) to $O(\log n)$ (where the number of operations grows **logarithmically**, making the algorithm much more efficient for large inputs).

7.4 Computing the Square Root (Heron's Method)

Heron's method (also known as the Babylonian method) is an ancient algorithm for computing approximations of square roots. The idea is to start with an estimate and progressively improve it.

Let's compute an approximation for $\sqrt{S}$. If $x_0 > 0$ is our initial estimate, we can improve our estimate using the following iterative formula:

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{S}{x_n} \right). \quad (7.1)$$

Let ε be the error in our estimate of $\sqrt{S}$. Then, $S = (x_0 + \varepsilon)^2$. Expanding the binomial, we get

$$S = (x_0 + \varepsilon)^2 = x_0^2 + 2x_0\varepsilon + \varepsilon^2. \quad (7.2)$$

We can solve the equation above for ε .

$$\varepsilon = \frac{S - x_0^2}{2x_0 + \varepsilon} \approx \frac{S - x_0^2}{2x_0}, \quad (\varepsilon \ll x_0). \quad (7.3)$$

Thus, we can compensate for the error and update our old estimate as

$$x_0 + \varepsilon \approx x_0 + \frac{S - x_0^2}{2x_0} = \frac{S + x_0^2}{2x_0} = \frac{\frac{S}{x_0} + x_0}{2} \equiv x_1 \quad (7.4)$$

Since the computed error was not exact, this is not the final answer, but it becomes our new estimate to use in the next iteration. The update process is repeated until the desired precision is achieved.

Let's implement this method recursively:

```
function raiz_quadrada(S, x_n = S / 2, ε = 0.0001)
    x_{n+1} = (x_n + S / x_n) / 2

    # Verificamos se a diferença entre as estimativas é menor que a precisão
    desejada
    if abs(x_{n+1} - x_n) < ε
        return x_{n+1}
    else
        return raiz_quadrada(S, x_{n+1}, ε)
    end
end

raiz_quadrada (generic function with 3 methods)
```

The function takes three parameters:

- S : the number whose square root we want
- x_n : our current estimate (by default, we start with half the number)
- ϵ : how close two consecutive estimates must be for us to consider that we've found the answer (by default, we're using 0.0001)

Let's test our function:

```
println(raiz_cuadrada(25)) # Deve ser próximo de 5
println(raiz_cuadrada(2))  # Deve ser próximo de 1.4142...
5.0000000000016778
1.4142135623746899
```

Notice what happens to the estimate's value when computing the square root of 25:

1. First call: $x_0 = 25/2 = 12.5$
2. Second call: $x_1 = (12.5 + 25/12.5)/2 = 7.25$
3. Third call: $x_2 = (7.25 + 25/7.25)/2 = 5.35$
4. Fourth call: $x_3 = (5.35 + 25/5.35)/2 = 5.01$
5. Fifth call: $x_4 = (5.01 + 25/5.01)/2 = 5.0$

7.5 Binomial Coefficient

The binomial coefficient $\text{binom } nk$ (read as "n choose k") represents the number of ways to choose k elements from a set of n elements, without regard to order. For example, $\text{binom } 52$ is the number of ways to choose 2 elements from a set of 5 elements.

The binomial coefficient has the following recursive definition (for $n > k$):

$$\text{binom } nk = \text{binom } n-1k-1 + \text{binom } n-1k \quad (7.5)$$

This formula can be derived by splitting the problem into two complementary cases. Consider a specific element X among the n available options. We can classify all possible subsets of k elements into two categories:

1. Subsets that include X : In this case, since X is already selected, we only need to choose $k-1$ additional elements from the remaining $n-1$ elements. This corresponds to $\text{binom } n-1k-1$ possibilities.
2. Subsets that do not include X : In this case, we must select all k elements from the remaining $n-1$ elements (excluding X). This corresponds to $\text{binom } n-1k$ possibilities.

The total number of possible subsets is the sum of these two cases, which justifies the recursive formula presented above.

We can determine the base cases from the following properties:

1. $\text{binom } n0 = 1$ for any $n \geq 0$ (there is only one way to choose 0 elements)
2. $\text{binom } nn = 1$ for any $n \geq 0$ (there is only one way to choose all the elements)

Let's implement this solution:

```
function coeficiente_binomial(n, k)
    if k == 0 || k == n
        return 1
    else
        return coeficiente_binomial(n - 1, k - 1) + coeficiente_binomial(n - 1, k)
    end
end
```

coeficiente_binomial (generic function with 1 method)

Let's test our function:

```
println(coeficiente_binomial(5, 2)) # Deve retornar 10
println(coeficiente_binomial(10, 4)) # Deve retornar 210
```

10
210

8. Automated Testing

We have developed several recursive algorithms, and to verify that they worked correctly, we ran manual tests by executing the functions with different inputs and checking their results. However, as our programs become more complex, this manual approach becomes inefficient and error-prone. In this chapter, we will introduce the concept of automated tests, which will let us check, systematically and reliably, whether our functions are behaving as expected.

8.1 Why Test?

When we write code, we want to be sure it is working correctly. Automated tests help us:

1. Check whether the code produces the expected results for different inputs
2. Catch bugs and errors before the program is used
3. Ensure that changes to the code do not break existing functionality
4. Document the expected behavior of our functions

In software development, a common practice is TDD (*Test-Driven Development*), where we first write the tests and then implement the code that satisfies them. This approach encourages a clearer design and a better understanding of the requirements even before we start programming.

8.2 Writing Tests With Conditionals

Let's start with a simpler approach to creating automated tests using conditional structures. We will consider the "Staircase Problem" that we saw in the previous chapter.

As a reminder, the problem consisted of determining how many different ways we can climb a staircase with n steps, if we can take steps of 1 or 2 steps at a time. Our solution was:

```
function maneiras_subir_escada(n)
  # Casos base
  if n == 0 || n == 1
    return 1
  else
    # Caso recursivo: soma das maneiras de chegar a partir de n-1 e n-2
    return maneiras_subir_escada(n - 1) + maneiras_subir_escada(n - 2)
  end
end
```

maneiras_subir_escada (generic function with 1 method)

Now, let's create a test function to check whether our implementation is correct:

```
function testa_maneiras_subir_escada()
  # Geralmente, verificamos alguns casos conhecidos ou que sabemos a resposta
  if maneiras_subir_escada(1) != 1
    println("Erro para n = 1")
    return false
  end

  if maneiras_subir_escada(2) != 2
    println("Erro para n = 2")
    return false
  end

  if maneiras_subir_escada(3) != 3
    println("Erro para n = 3")
    return false
  end

  if maneiras_subir_escada(4) != 5
```

```

        println("Erro para n = 4")
        return false
    end

    println("Todos os testes para a função maneiras_subir_escada passaram!")
    return true
end

# Executamos os testes
testa_maneiras_subir_escada()

Todos os testes para a função maneiras_subir_escada passaram!

true

```

In this test function, we check whether our implementation returns the correct values for different inputs. If a test fails, we display a message indicating which case failed. If all tests pass, we display a success message.

This is an important principle for automated tests: **if the test passes, it should only indicate that it succeeded!** This means that, ideally, tests should not print many messages when everything is working correctly, only when something goes wrong.

Let's do the same for the "Binomial Coefficient" calculation, which we also saw in the previous chapter:

```

function coeficiente_binomial(n, k)
    if k == 0 || k == n
        return 1
    else
        return coeficiente_binomial(n - 1, k - 1) + coeficiente_binomial(n - 1, k)
    end
end

function testa_coeficiente_binomial()
    if coeficiente_binomial(5, 2) != 10
        println("Erro para (5, 2)")
        return false
    end

    if coeficiente_binomial(10, 4) != 210
        println("Erro para (10, 4)")
        return false
    end

    if coeficiente_binomial(7, 3) != 35
        println("Erro para (7, 3)")
        return false
    end

    println("Todos os testes para a função coeficiente_binomial passaram!")
    return true
end

# Executamos os testes
testa_coeficiente_binomial()

Todos os testes para a função coeficiente_binomial passaram!

true

```

8.3 Testing With the Test Module

So far, we have created test functions manually using conditional structures. However, Julia provides a built-in testing module called `Test`, which offers more advanced functionality for automated tests.

Let's rewrite our tests using the `Test` module:

```
using Test

@testset "Testes para maneiras_subir_escada" begin
    @test maneiras_subir_escada(1) == 1
    @test maneiras_subir_escada(2) == 2
    @test maneiras_subir_escada(3) == 3
    @test maneiras_subir_escada(4) == 5
end

@testset "Testes para coeficiente_binomial" begin
    @test coeficiente_binomial(5, 2) == 10
    @test coeficiente_binomial(10, 4) == 210
    @test coeficiente_binomial(7, 3) == 35
end
```

```
Test Summary:                               | Pass
Total Time
Testes para maneiras_subir_escada |    4        4  0.4s
Test Summary:                       | Pass Total Time
Testes para coeficiente_binomial |    3        3  0.0s
```

```
Test.DefaultTestSet("Testes para coeficiente_binomial", Any[], 3, false, false,
true, 1.789736803233554e9, 1.789736803233596e9, false, "In[4]")
```

With the `Test` module, we use the `@testset` macro to group related tests and the `@test` macro to check specific conditions. If a test fails, the module automatically displays useful information about the failure, such as the expression that failed and the expected versus obtained values.

In addition, the `Test` module offers other useful macros:

- `@test_throws`: checks whether an expression throws a specific exception
- `@test_approx_eq`: checks whether two floating-point values are approximately equal (accounting for rounding errors)
- `@test_broken`: marks a test that is expected to fail (useful for documenting known bugs)

8.4 More Examples

Let's implement two new functions and their respective tests: one function to calculate the sum of the digits of a number, and another to check whether a number is prime.

8.5 Sum of Digits

First, let's create a function that calculates the sum of the digits of an integer. For example, for the number 123, the sum of the digits would be $1 + 2 + 3 = 6$.

Before implementing the function, let's think about the test cases:

- If the function receives a single-digit integer, it should return that digit
- If the function receives 100, it should return $1 + 0 + 0 = 1$
- If the function receives 123, it should return $1 + 2 + 3 = 6$
- If the function receives 99, it should return $9 + 9 = 18$

We can implement the function using recursion. The idea is to “peel” the number, extracting one digit at a time:

```
function soma_digitos(n)
  if n <= 0
    return 0
  else
    # Obtemos o último dígito com o resto da divisão por 10
    ultimo_digito = n % 10
    # Removemos o último dígito com a divisão inteira por 10
    resto_numero = n ÷ 10
    # Somamos o último dígito com a soma dos dígitos do resto do número
    return ultimo_digito + soma_digitos(resto_numero)
  end
end
```

soma_digitos (generic function with 1 method)

The test cases discussed above can be implemented using the Test module:

```
@testset "Testes para soma_digitos" begin
  @test soma_digitos(0) == 0
  @test soma_digitos(1) == 1
  @test soma_digitos(100) == 1
  @test soma_digitos(123) == 6
  @test soma_digitos(99) == 18
end
```

```
Test Summary: | Pass Total Time
Testes para soma_digitos | 5 5 0.0s
```

```
Test.DefaultTestSet("Testes para soma_digitos", Any[], 5, false, false, true,
1.78973680338689e9, 1.789736803390929e9, false, "In[6]")
```

8.6 Checking for Prime Numbers

Let’s create a function to check whether a number is prime. A prime number is one that is divisible only by 1 and by itself. Before writing the function, let’s think about the tests:

- By definition, any number less than or equal to 1 is not prime
- The number 2 is prime (easy to check)
- The number 3 is prime (also easy to check)
- The number 4 is not prime, since 2 also divides 4
- The number 17 is prime
- The number 25 is not prime, since 5 also divides 25

We can implement the function using a recursive approach that tries to divide the number by each integer from 2 up to the square root of the number:

```
function verifica_divisor(n, divisor)
  # Se encontramos um divisor, o número não é primo
  if n % divisor == 0
    return false
  # Se já testamos até a raiz quadrada, o número é primo
  elseif divisor * divisor > n
    return true
  else
    # Continua verificando com o próximo divisor
    return verifica_divisor(n, divisor + 1)
  end
end
```

```
function e_primo(n)
    if n <= 1 # Por definição
        return false
    elseif n == 2 # Primeiro primo
        return true
    else
        # Verifica se n tem algum divisor começando com 2
        return verifica_divisor(n, 2)
    end
end

e_primo (generic function with 1 method)
```

The tests can be written as:

```
@testset "Testes para e_primo" begin
    @test e_primo(2) == true
    @test e_primo(3) == true
    @test e_primo(4) == false
    @test e_primo(17) == true
    @test e_primo(25) == false
end
```

```
Test Summary:          | Pass  Total  Time
Testes para e_primo |     5      5  0.0s
```

```
Test.DefaultTestSet("Testes para e_primo", Any[], 5, false, false, true,
1.7897368034724e9, 1.789736803488476e9, false, "In[8]")
```

8.7 Check Your Understanding

1. What is the difference between using conditional structures and the Test module for automated testing?
2. Why are automated tests important in software development?

8.8 Explore on Your Own

1. Explore other macros available in Julia's Test module and try using them in your own tests.

9. Iterative Repetition Structures

So far, we've seen that computers are very good at performing calculations and repetitions. We carried out these repetitions using recursive functions, where a function calls itself to solve smaller problems. Now we'll look at an alternative way of performing repetitions: the iterative approach.

9.1 Introduction to while

The while structure is one of the most fundamental ways of creating loops in programming. It allows a block of code to be executed repeatedly while a specific condition is true, in an iterative fashion. The basic syntax of while in Julia is:

```
while condição
    # Bloco de código a ser repetido
end
```

The while structure works through these steps:

1. The condition is evaluated
2. If the condition is true, the block of code is executed
3. After the block runs, the condition is evaluated again
4. This cycle continues until the condition becomes false

An important thing to understand about while is that, to avoid an infinite loop (a loop that never ends), something related to the condition must be changed inside the block of code.

Let's start with a simple example: a countdown.

```
function contagem_regressiva(n)
    while n > 0
        println(n)
        n = n - 1 # Esta linha é essencial para evitar um loop infinito
    end
    println("Fim!")
end

contagem_regressiva(5)

5
4
3
2
1
Fim!
```

In this example, the condition `n > 0` is initially true (assuming `n` starts with a positive value). The block of code prints the current value of `n` and then decrements `n` by 1. Eventually, `n` will reach zero, making the condition false and ending the loop.

9.2 Comparing Recursion and Iteration

To better understand the difference between recursion and iteration, let's rewrite some functions we previously implemented using recursion.

9.3 Countdown

First, let's recall the recursive version of the countdown:

```
function contagem_recursiva(n)
    if n <= 0
        println("Fim!")
    end
end
```

```

        else
            println(n)
            contagem_rekursiva(n - 1)
        end
    end
end

contagem_rekursiva(5)
5
4
3
2
1
Fim!

```

Comparing the two implementations, we can observe that:

- In the recursive version, the base case ($n \leq 0$) corresponds to the while loop's stopping condition
- The recursive call with $n - 1$ corresponds to the update of n in the while loop

Both versions produce the same result, but with different approaches.

9.4 Sum of the First N Numbers

Let's implement a function that calculates the sum of the first n positive integers ($1 + 2 + \dots + n$), using both recursion and while.

Recursive version:

```

function soma_rekursiva(n)
    if n <= 0
        return 0
    else
        return n + soma_rekursiva(n - 1)
    end
end

println("Soma dos primeiros 5 números (recursiva): ", soma_rekursiva(5))
Soma dos primeiros 5 números (recursiva): 15

```

Version with while:

```

function soma_while(n)
    soma = 0
    i = 1

    while i <= n
        soma = soma + i
        i = i + 1
    end

    return soma
end

println("Soma dos primeiros 5 números (while): ", soma_while(5))
Soma dos primeiros 5 números (while): 15

```

In the recursive version, we have an explicit base case ($n \leq 0$) and a recursive call that reduces the problem. In the while version, we use a control variable i that is incremented on each iteration, and an accumulator variable $soma$ that stores the partial result.

9.5 Calculating Mathematical Series

Repetition structures are especially useful for calculating sums of mathematical series. Let's implement a function to calculate the approximation of sine using the Taylor series:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \quad (9.1)$$

This series can be represented as:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n+1}}{(2n+1)!} \quad (9.2)$$

Implementation using while:

```
function sin_taylor(x, termos = 10)
    resultado = 0.0
    termo = x
    i = 0

    while i < termos
        # Adicionamos o termo atual à soma
        resultado = resultado + termo

        # Calculamos o próximo termo
        i = i + 1
        termo = -termo * x * x / ((2 * i) * (2 * i + 1))
    end

    return resultado
end

# Teste com π/6 (30 graus), cujo seno é 0.5
println("sin(π/6) ≈ ", sin_taylor(π/6))
println("sin(π/6) exato: ", sin(π/6))

sin(π/6) ≈ 0.49999999999999994
```

```
sin(π/6) exato:
0.49999999999999994
```

Notice how while allows precise control over the number of terms of the series we want to calculate.

Let's compare this with a recursive implementation:

```
function sin_taylor_recursivo(x, i = 0, termos = 10, termo = x, resultado = 0.0)
    if i >= termos
        return resultado
    else
        # Adicionamos o termo atual à soma
        novo_resultado = resultado + termo

        # Calculamos o próximo termo
        novo_i = i + 1
        novo_termo = -termo * x * x / ((2 * novo_i) * (2 * novo_i + 1))
    end
end
```

```

        return sin_taylor_recursivo(x, novo_i, termos, novo_termo, novo_resultado)
    end
end

println("sin( $\pi/6$ ) recursivo  $\approx$  ", sin_taylor_recursivo( $\pi/6$ ))
sin( $\pi/6$ ) recursivo  $\approx$  0.49999999999999994

```

The recursive version is more complex here, since it needs several additional parameters to maintain state between recursive calls. The while version is clearer and more straightforward in this case.

9.6 When to Use Recursion and Iteration?

Both recursion and iteration can be used to solve repetition problems, each with its own strengths:

Advantages of iteration

- Generally more memory efficient
- Avoids the risk of stack overflow for large inputs
- Can be more intuitive for simple repetition operations
- Allows more detailed control over the iteration process

Advantages of recursion

- Often more elegant for problems that decompose naturally
- Can make code more concise and readable for certain algorithms
- Directly reflects recursive mathematical definitions
- Particularly useful for hierarchical data structures

A practical rule of thumb is:

- Use iteration when you need to repeat an operation a fixed or indeterminate number of times
- Use recursion when the problem can naturally be divided into smaller subproblems of the same type

9.7 Check Your Understanding

1. What is the difference between using recursion and iteration for repetition?
2. Implement a function that counts the number of digits in an integer using the while structure.
3. Given an integer, write a function that reverses its digits (for example, 123 would become 321).

9.8 Explore on Your Own

1. Research loop optimization (loop unrolling) and try applying this concept to a function using while.

10. Exercise Class

10.1 Revisiting Factorial: Recursive and Iterative

Now that we've also learned how to do repetitions with the while command, it's always good to think about which command is most appropriate. Let's look at the example below with two versions of the function for calculating factorial.

```
function fatorial_recursivo(n::Int64) # Com o ::Int64 estamos definindo que o
# Caso base do fatorial: 0! e 1! são iguais a 1
    if n == 0 || n == 1
        return 1
    # Chamada recursiva: n! = n * (n-1)!
    else
        return n * fatorial_recursivo(n - 1)
    end
end

function fatorial_iterativo(n::Int64)
    # Inicializa o resultado como 1 (já que o fatorial de 0 é 1)
    resultado = 1

    # No loop estamos fazendo a multiplicação: n * (n-1) * ... * 2
    while n > 1
        # Multiplica o resultado pelo valor atual de n
        resultado *= n

        # Decrementa n em 1 para continuar o cálculo do fatorial
        n -= 1
    end
    return resultado
end

println(fatorial_recursivo(3))
6
```

There's something new in the code above: the function parameters declare the type explicitly. In this case, we're saying that the value *n* the function will receive is of a specific type. That is, a 64-bit integer.

The coding style is a bit different from before, since it was written by someone else, the teaching assistant. We can see that she has the habit of using longer variable names, as well as using contractions like += and -=.

10.2 Approximating the Square Root

For the next example, let's look at the Newton-Raphson method for calculating a square root. It's a recursive method in which the next value is based on the previous value. The more calls that are made, the closer we get to the final value.

More information about the method can be found [here](#). But for now, let's think about the following implementation. To calculate the root, we can use the following formula, starting from an initial guess *r*, for the root of *x*.

$$r_{n+1} = 0.5 * (r + x/r) \quad (10.1)$$

Since the code below is more complicated, it includes comments.

```

function aproxima_raiz(x::Float64, epsilon::Float64)::Float64
    if x < 0
        return nothing
    end

    # Chute inicial
    aproximacao = x/2
    melhor_aproximacao = aproximacao

    while true
        # Fórmula para aproximação de raiz quadrada utilizando o método de
        # Newton-Raphson
        melhor_aproximacao = 0.5 * (aproximacao + x/aproximacao)

        # Se a distância absoluta entre os dois pontos é menor do que epsilon,
        # então podemos parar o método
        if abs(aproximacao - melhor_aproximacao) <= epsilon
            break
        end

        # Se a aproximação ainda não for boa o suficiente, então atualizamos a
        # aproximação para a próxima iteração
        aproximacao = melhor_aproximacao
    end

    return melhor_aproximacao
end

```

aproxima_raiz (generic function with 1 method)

Notice that a new command was introduced, break. This command interrupts the execution of the while loop, forcing an exit from it.

10.3 Checking Whether a Number Is Prime

In the next example, we'll check whether a number is prime, that is, whether its only divisors are 1 and itself. The simplest way to do this is to try dividing the number by others. If any of them divides it, the number is not prime.

```

function verifica_primo(num :: Int64)
    if num <= 1
        return false
    end
    i=2
    # pode ser melhorado com i<=num/2
    # ou também com i<= sqrt(num): baseado no fato que um número composto deve ter
    # um fator menor ou igual a raiz desse número
    while i<num
        if num % i == 0
            return false
        end
        i+=1
    end
    return true
end

```

verifica_primo (generic function with 1 method)

Just as the break command is used to interrupt the execution of a loop, the return command can be used to end the execution of a function at any moment.

10.4 Checking Whether a Number Is a Palindrome

A palindromic number is a number that is symmetric. That is, reading the digits from left to right gives the same result as reading the digits in reverse order. For example, the number 121 is a palindrome, as are 11 and 25677652. Single-digit numbers are also palindromes.

```
function e_palindromo(n::Int64)
    #=
        Guarda os dígitos de n que ainda devem ser invertidos
        A variável auxiliar é necessária para que o valor de n não seja, perdido,
        e possamos usar ele posteriormente.
    =#
    aux = n
    # Guarda a inversão do número n
    n_inv = 0

    #=
        Continuamos o while enquanto ainda há números a serem invertidos,
        ou seja, enquanto aux for maior que 0.
    =#
    while aux > 0
        # Coloca o último dígito de aux na variável que guarda a inversão
        resto = aux % 10
        n_inv = n_inv * 10 + resto

        # Retira o último dígito de aux
        aux = div(aux, 10)
    end

    if n == n_inv
        println("O número $n é palíndromo")
    else
        println("O número $n não é palíndromo")
    end
end

e_palindromo(2002)
e_palindromo(1234)

0 número 2002 é palíndromo
0 número 1234 não é palíndromo
```


11. Revisiting the Previous Class

In this class, in addition to reviewing what we saw last time, we looked at a new solution to the problem of checking whether a number is a palindrome.

For this, we used a slightly different technique: instead of reversing the number and comparing it to the original, we check whether its outer digits are equal.

Consider the number 234432. The first step is to check that at the extremes, the most significant and least significant digits, we have the number 2. Next, we can continue the check with the number 3443. If the check ever fails, the number is not a palindrome.

The tests and code follow below.

```
using Test

function testaPal()
    @test testaPal(1)
    @test testaPal(131)
    @test testaPal(22)
    @test testaPal(53877835)
    @test !testaPal(123)
    @test !testaPal(23452)
    println("Final dos testes")
end

function testaPal(n::Int64)
    # o primeiro passo é encontrar um número com o mesmo número de dígitos de n
    pot10 = 1
    while pot10 < n
        pot10 = pot10 * 10
    end
    pot10 = div(pot10, 10)

    while n > 9
        d1 = n % 10
        d2 = div(n, pot10)
        if d1 != d2
            return false
        end
        n = div(n % pot10, 10)
        pot10 = div(pot10, 100)
    end
    return true
end

testaPal (generic function with 2 methods)
```

11.1 Randomness

In Julia we have the `rand()` function, which returns a floating-point number between 0 and 1. Depending on the parameters, we can get other types of numbers, such as:

```
rand{Int}() # devolve um inteiro
rand{Int, Int}() # devolve um número entre 1 e 10
rand{Bool}() # devolve verdadeiro ou falso

true
```

But before looking at code that uses `rand()`, let's think about a real-world problem. Imagine we need to run a fair drawing, and the only instrument we have for it is a biased coin, one that lands on heads far more often than tails. Can we use this coin for a fair drawing?

The idea for solving this problem is to look at pairs of flips. That is, we ignore pairs where we get two heads or two tails. In the remaining pairs, we'll have one heads and one tails, or vice versa. The chances of either outcome will be 50%. This way, we can correct for the biased coin.

To simplify the exercise, the coin can return 0 or 1, corresponding to heads or tails. Look at the following function that simulates a biased coin.

```
function sorteio()  
  if rand() > 0.90  
    return 1  
  else  
    return 0  
  end  
end
```

sorteio (generic function with 1 method)

Notice that the function returns 0 most of the time. We can even verify this by running a thousand flips:

```
function verificaSorteio()  
  cara = 0  
  coroa = 0  
  i = 0  
  while i < 1000  
    if sorteio() == 0  
      cara = cara + 1  
    else  
      coroa = coroa + 1  
    end  
    i = i + 1  
  end  
  println("O número de caras foi: ", cara, " e de coroas foi :", coroa)  
end
```

verificaSorteio (generic function with 1 method)

But we can correct the flip as follows:

```
function sorteioBom()  
  sorteio1 = sorteio()  
  sorteio2 = sorteio()  
  while sorteio1 == sorteio2 # se forem iguais, tente novamente  
    sorteio1 = sorteio()  
    sorteio2 = sorteio()  
  end  
  return sorteio1 # ao termos um diferente, podemos devolver o primeiro sorteio  
end
```

sorteioBom (generic function with 1 method)

We can use `verificaSorteio` to see the difference.

```
function verificaSorteio()  
  cara = 0  
  coroa = 0  
  i = 0  
  while i < 1000
```

```

    if sorteioBom() == 0
        cara = cara + 1
    else
        coroa = coroa + 1
    end
    i = i + 1
end
println("O número de caras foi: ", cara, " e de coroas foi : ", coroa)
end

```

verificaSorteio (generic function with 1 method)

We can also approximate Euler's number (e), the mathematical constant that is the base of the natural logarithm, using a probabilistic simulation. The idea behind this code is that the average number of attempts needed for the sum of random numbers between 0 and 1 to exceed 1 approaches the value of e .

```

function calculaEuler(total)
    soma_tentativas = 0
    for i in 1:total
        soma = 0.0
        tentativas = 0
        while soma <= 1 # Continue gerando números até a soma ultrapassar 1
            soma += rand() # Gera número aleatório entre 0 e 1
            tentativas += 1
        end
        soma_tentativas += tentativas # Somar o número de tentativas
        necessárias
    end
    return soma_tentativas / total # A média do número de tentativas será uma
    estimativa de e
end

```

```

println("Estimativa de e (1000 iterações): ", calculaEuler(1000))
println("Estimativa de e (100000 iterações): ", calculaEuler(100000))
println("Estimativa de e (100000000 iterações): ", calculaEuler(100000000))

```

Estimativa de e (1000 iterações): 2.728

Estimativa de e (100000 iterações): 2.7237

Estimativa de e (100000000 iterações): 2.7181607

To wrap up the class, let's apply the Monte Carlo method to calculate Pi. Picture the first quadrant, where we have a quarter-circle of radius 1 inside a square with side 1. We can draw random points, and those that fall inside the circle count toward its area. More information can be found here (https://pt.wikipedia.org/wiki/Método_de_Monte_Carlo)

```

function calculaPi(total)
    noAlvo = 0
    i = 0
    while i < total
        x = rand() / 2.0 # gera um número entre 0 e 0.5
        y = rand() / 2.0
        if sqrt(x * x + y * y) <= 0.5
            noAlvo = noAlvo + 1
        end
        i = i + 1
    end
    return 4 * (noAlvo / total) # precisamos multiplicar para ter a área de 4

```

```
quadrantes  
end
```

```
println(calculaPi(100))  
println(calculaPi(1000000))  
println(calculaPi(1000000000))
```

2.92

3.141012

3.141642808

12. Data Input and the Beginning of Lists

In this lesson, we have two main topics: how to perform data input, through input commands and through command-line arguments. We will also see how to work with a special type of variable, one that can store more than one value.

12.1 The Input Command

When we want to insert data in Julia, we can simply enter the data. But how can we get a regular program to accept data as input?

For this, we have the `readline()` command, which pauses the program's execution and waits for a String to be entered, which happens when the "enter" key is pressed.

```
println("Digite o seu nome")
resposta = readline()
println("O seu nome é: ", resposta)
```

If, when running the program, you type Maria and press enter, the final output of your program will be O seu nome é: Maria.

Since `readline()` reads Strings, if we want to read numbers, we need to use the `parse` command. In its simple form the `parse` command takes two parameters: the first is the type we want to convert to, and the second is the original value.

```
println("Digite um inteiro")
valor = parse{Int64, readline()}
println("O numero digitado foi ", valor)
```

Now that we know how to read numbers from the keyboard, let's try a simple exercise: read a sequence of integers terminated by zero and return their sum.

```
function somaVarios()
    soma = 0.0
    println("Digite um número")
    n = parse{Float64, readline()}
    while n != 0
        soma = soma + n
        println("Digite um número")
        n = parse{Float64, readline()}
    end
    println("A soma é: ", soma)
end
```

Look at the following example, which calculates the squares of the numbers in a list terminated by zero.

```
function leQ()
    x = readline()
    n = parse{Float64, x}
    while n != 0
        println("$n ao quadrado é ", n * n)
        x = readline()
        n = parse{Float64, x}
    end
end
```

Note that `readline` can also take a file variable so that data is read directly from it. In this case, however, we need to be careful to open (`open()`) and close (`close()`) the file, as shown below:

```
function leQ()
    println("Digite um número")
    f = open("numeros.txt", "r+")
    x = readline(f)
    n = parse(Float64, x)
    while n != 0
        println("$n ao quadrado é ", n * n)
        println("Digite outro número")
        x = readline(f)
        n = parse(Float64, x)
    end
    close(f)
end
```

12.2 Reading Through the Command Line

The other way to read input is through the `ARGS` constant, which is set up when a program is called. To understand this better, let's look at the following program.

```
println(ARGS)
```

If the line above is in the file `args.jl`, calling `julia args.jl` with different parameters will produce different results.

For example, when calling:

```
julia args.jl 1 2 3 abc
```

We will get the following response:

```
["1", "2", "3", "abc"]
```

Let's take a closer look at this response, noting that each parameter occupies a position.

```
tam = length(ARGS)
println("O tamanho dos argumentos é: ", tam)
for i in 1:tam
    println(ARGS[i])
end
```

Looking at the code above, we can see that the `length()` function returns the number of arguments, that is, the size of the `ARGS` list. In addition, using square brackets we can access each position of the list individually.

The example below adds up the integer parameters given as arguments. It also illustrates a good practice: always organizing code into modules, in this case into functions:

```
function SomaEntrada()
    tam = length(ARGS)
    s = 0
    i = 1
    while i <= tam
        valor = parse{Int, ARGS[i]}
        println(valor)
    end
end
```

```

        s = s + valor
        i = i + 1
    end
    println("A soma foi: ", s)
end
SomaEntrada()

```

Lists give us a lot of flexibility. For this reason, lists, or arrays, deserve a topic of their own.

12.3 Lists

Let's first play around a bit in the console.

```

vetor = [1, 2, 3]
println(vetor[1])
println(length(vetor))
vetor[2] = vetor[2] + 1
vetor[1] = 2 * vetor[3]
println(vetor)
1
3
[6, 3, 3]

```

As mentioned before, the for loop was made to work with arrays. Let's look at a few functions, the first one prints the elements of an array one per line.

```

function imprimeVetor(v)
    for el in v
        println(el)
    end
end

```

This can also be done using the array's indices:

```

function imprimeVetor(v)
    for i in 1:length(v)
        println(v[i])
    end
end

```

Since each position is independent, we can calculate the sum of the odd elements of an array

```

function somaImpVetor(v)
    soma = 0
    for i in 1:length(v)
        if v[i] % 2 == 1
            soma = soma + v[i]
        end
    end
    return soma
end

```

We also saw a few other examples in class, such as calculating the average of the elements in an array.

```

function mediaV(v)

```

```

soma = 0.0
for i in v
    soma = soma + i
end
return soma / length(v)
end

```

Return the sum of the odd elements of an array

```

function somaImpar(v)
    soma = 0
    for i in v
        if i % 2 == 1
            soma = soma + i
        end
    end
    return soma
end

```

Print the numbers in an array that are divisible by 5.

```

function imprimeDivisivelPor5(v)
    for i in v
        if i % 5 == 0
            println(i)
        end
    end
end

```

With a small variation, using the push!() command, we can see how to return an array with the numbers divisible by 5.

```

function devolveDivisivelPor5(v)
    x = [] # começa com um vetor vazio
    for i in v
        if i % 5 == 0
            push!(x, i) # adiciona um elemento ao vetor x
        end
    end
    return x
end

```

12.4 Linear Algebra and Lists

Manipulating lists is a fundamental part of linear algebra, which studies vectors and matrices. Functions like the dot product of two vectors are classic examples. Below are two examples of the dot product of two vectors. Recall that it is defined as the sum of the products of elements in matching positions.

```

function dotProduct(a, b)
    soma = 0
    if length(a) != length(b)
        return soma # o produto não está definido se os tamanhos são diferentes
    end
    for i in 1:length(a)
        soma = soma + a[i] * b[i]
    end
end

```

```

    return soma
end

```

Above, we saw that a special case of using for consists of making the for loop vary between 1 and a size (1:length(a))

Notice the difference in the version below:

```

function dotProduct(a, b)
    soma = 0
    if length(a) != length(b)
        return soma # o produto não está definido se os tamanhos são diferentes
    end
    i = 1
    for x in a
        soma = soma + x * b[i]
        i = i + 1
    end
    return soma
end

```

12.5 Permutation Exercise

To finish, let's write a function that, given a vector of integers of size n , checks whether that vector is a permutation of the numbers from 1 to n . To do this, we will check whether each number from 1 to n is in the vector.

But, without forgetting the tests:

```

@testset "Verifica Permutação" begin
    @test permuta([1,2,3])
    @test permuta([3, 2, 1])
    @test permuta([1])
    @test permuta([2, 1])
    @test permuta([4, 2, 3, 1])
    @test !permuta([1, 1])
    @test !permuta([1, 3])
    @test permuta([])
end

```

and the code:

```

function permuta(v)
    tam = length(v)
    for i in 1:tam
        if !(i in v)
            return false
        end
    end
    return true
end

```

We used Julia's `in` command, which checks whether an element is in the vector.

13. Vector Exercises

Vectors let us implement much more complex algorithms. In this chapter, we'll look at a few exercises.

13.1 Permutation

Given a vector of integers, we want to check whether it contains a permutation. To do this, we need to check whether a vector of size n contains the numbers from 1 to n , each appearing exactly once. The vector $[3, 1, 2]$ is a permutation, since it has size 3 and the elements from 1 to 3 each appear once.

One way to solve this problem is with a flag variable. We start by assuming the vector is a permutation, then check whether all the numbers between 1 and n are in the vector. This can be done with the `in` command, which checks whether an element belongs to the vector.

```
function permutação(l)
    perm = true
    tamanho = length(l)
    i = 1
    while i <= tamanho
        if !(i in l)
            perm = false
        end
        i += 1
    end
    return perm
end
```

permutação (generic function with 1 method)

Another alternative is to check, for each element of the vector, whether it is between 1 and n and unique. In other words, we check whether the first element is between 1 and n , and then scan the vector to see if it's unique. Then we do the same for the following elements. The code looks like this:

```
function permutação(l)
    perm = true
    tamanho = length(l)
    i = 1
    while i <= tamanho
        if (l[i] > tamanho || l[i] <= 0)
            perm = false
        end
        j = i + 1
        while j <= tamanho
            if l[j] == l[i]
                perm = false
            end
            j += 1
        end
        i += 1
    end
    return perm
end
```

permutação (generic function with 1 method)

Yet another alternative is to use an auxiliary vector where we count the occurrences of each number between 1 and n . At the end, every element of this auxiliary vector must equal 1. This time, we'll take the opportunity to add automated tests as well.

```

using Test
function permutação(l)
    perm = true
    tamanho = length(l)
    aux = zeros{Int8, tamanho}
    for i in l
        if i < 1 || i > tamanho
            perm = false
        else
            aux[i] += 1
        end
    end
    for i in aux
        if i != 1
            perm = false
        end
    end
    return perm
end

@testset "Verifica Permutação" begin
    @test permutação([1,2,3])
    @test permutação([3, 2, 1])
    @test permutação([1])
    @test permutação([2, 1])
    @test permutação([4, 2, 3, 1])
    @test !permutação([1, 1])
    @test !permutação([1, 3])
    @test !permutação([4, 2, 3, -1])
    @test !permutação([5, 2, 3, 1])
    @test permutação([])
    @test !permutação([0, 3, 3])
    @test !permutação([2, 2, 2])
end

```

Test Summary: | Pass

Total Time

Verifica Permutação | 12 12 0.5s

Test.DefaultTestSet("Verifica Permutação", Any[], 12, false, false, true, 1.789736806066159e9, 1.7897368065552e9, false, "In[3]")

13.2 Histogram

Since we saw in the previous example how to “count” numbers, we can take it a step further and compute the histogram of a vector containing numbers between 1 and 10.

```

using Test

function histograma(l)
    result = [0,0,0,0,0,0,0,0,0,0]
    i = 1
    while i <= length(l)
        valor_atual = l[i]
        if valor_atual >= 1 && valor_atual <= 10
            result[valor_atual] += 1
        end
        i += 1
    end
    return result
end

```

```

@testset "Verifica Histograma" begin
    @test [1,0,0,0,0,0,0,0,0,0] == histograma([1])
    @test [0,0,0,0,0,0,0,0,0,0] == histograma([-1])
    @test [0,0,1,0,0,0,0,0,0,0] == histograma([3])
    @test [0,0,0,0,0,0,0,0,0,1] == histograma([10])
    @test [0,0,0,0,0,0,0,0,0,0] == histograma([11])
    @test [1,4,0,2,5,1,0,1,0,0] == histograma([5,6,5,4,5,5,4,2,8,2,1,2,5,2])
    @test [0,0,0,0,0,0,0,0,0,0] == histograma([])
end

```

```

Test Summary:      | Pass  Total  Time
Verifica Histograma |    7      7  0.1s

```

```

Test.DefaultTestSet("Verifica Histograma", Any[], 7, false, false, true,
1.789736806702366e9, 1.789736806766942e9, false, "In[4]")

```

13.3 Modeling Problems with the Computer

The computer can be a powerful tool for modeling real-world problems. Let's take the birthday problem as an example. This problem is also known as the birthday paradox: calculate the probability that, in a room with n people, at least two of them share the same birthday. This problem can be solved using probability theory, which reveals that in a room of 23 people, the chance that two of them share a birthday is just over 50%.

But we can also model this problem computationally. The first step is to simplify the dates: instead of month and year, we can encode each day as a number between 1 and 365, where 1 corresponds to January 1st. To solve the problem, we can randomly draw n dates and check whether there's any repetition; if there is, we've found two people with the same birthday.

This is represented in the `experimento_niver` function below. But to find out the actual probability, we need to repeat the experiment many times. In the `main()` function below, we ask for the number of experiments and the number of people in order to run the simulation.

```

function experimento_niver(n)
    repetiu = false
    i = 1
    nivers = []
    while i <= n && (repetiu == false)
        niver = rand(1:365)
        if niver in nivers
            repetiu = true
        end
        push!(nivers, niver)
        i += 1
    end
    return repetiu
end

```

```

function main()
    print("Quantos experimentos? ")
    quantas = readline()
    print("Quantas pessoas? ")
    npessoas = readline()
    quantas = parse{Int64}(quantas)
    npessoas = parse{Int64}(npessoas)
    sucessos = 0
    i = 1
    while i <= quantas
        if experimento_niver(npessoas)
            sucessos += 1
        end
        i += 1
    end
end

```

```

        end
        i += 1
    end
    println("A probabilidade estimada é ", 100*sucessos/quantas, "%")
end
main()

```

With small variations, we can run other experiments, such as checking whether more than two people share the same birthday. To do this, below we count the number of repetitions.

```

function experimento_niver(n)
    repetiu = 0
    i = 1
    nivers = []
    while i <= n
        niver = rand(1:365)
        if niver in nivers
            repetiu += 1
        end
        push!(nivers, niver)
        i += 1
    end
    return repetiu >= 2
end
experimento_niver (generic function with 1 method)

```


14. Modeling Blackjack with Programming

In this chapter, we will build a simplified terminal version of the Blackjack game and use computational simulations to evaluate different playing strategies. Blackjack (also known as “21”) is a card game where the goal is to get closer to 21 points than your opponent, without going over that value.

This project will let us practice important programming concepts while exploring how simulation can help solve real-world problems.

14.1 Interactive Version of the Game

The first step, before thinking about the implementation, is to understand what we want to model. Blackjack is played by two people: the player and the *dealer* (the house or casino). In our game, the dealer will be the computer. In addition, we will follow these game rules:

1. Initially, the player and the dealer each receive two cards.
2. During the player’s turn, the player can choose between two options: draw another card (*hit*) or stop drawing cards (*stand*).
3. During the dealer’s turn, the dealer will follow this strategy: keep drawing cards while the total is below 17, and stop otherwise.

The player’s goal is to get closer to 21 than the dealer, without going over that value.

The value of each card matches its face value. The Queen (Q), Jack (J), and King (K) cards are worth 10 points. The Ace can be worth 1 or 11 (depending on the player’s strategy). We will treat an Ace as being worth 11, unless that would cause the hand to go over 21 (bust). In that case, the Ace is worth 1. Now that we know the rules of the game, we can implement it.

Let’s start by creating a deck of 52 cards. A simple way to represent it is with a vector (we’ll call it *baralho*, meaning “deck”) where each element represents a card type and stores how many of that type are still available, meaning that if *baralho[i]* is 4, there are 4 cards of type *i* available in the deck.

For the numeric representation of the cards, the Ace corresponds to 1, the Two to 2, and so on up to 10. The Queen is represented by 11, the Jack by 12, and the King by 13. Since the suit doesn’t matter, we consider that there are four cards of each type in a full deck.

```
function criaBaralho()
    baralho = zeros(Int8, 13)

    i = 1
    while i < 14
        baralho[i] = 4
        i = i + 1
    end

    return baralho
end
```

criaBaralho (generic function with 1 method)

Besides creating the deck, we need a function to draw a card from it. In real life, the deck would be shuffled with the cards face down, and drawing a card would just mean pulling the one on top. The problem is that our model doesn’t support that kind of behavior, since the vector only represents the count of cards of each type.

To simulate this behavior, we can draw a random number between 1 and 13 and check whether cards of that type still exist. If they do, we remove one card of that type from the deck and return

the drawn number. Otherwise, we keep drawing numbers until we find a type that still has cards available.

```
function pegaCarta(baralho)
  i = rand(1:13)

  # Verifica se existem cartas do tipo sorteado
  while baralho[i] == 0
    i = rand(1:13)
  end

  baralho[i] = baralho[i] - 1

  return i
end
```

pegaCarta (generic function with 1 method)

Now that we have a deck and can draw cards from it, we need a function to calculate the value of a hand. To represent the player's and dealer's hands, we will use a vector that stores the values of the cards received. Both hands will be defined in a function later on.

```
function calculaValorMao(mao)
  ases = 0
  valor = 0
  tamanhoMao = length(mao)

  i = 1
  while i <= tamanhoMao
    if mao[i] == 1 # Ás
      ases = ases + 1
      valor = valor + 11
    elseif mao[i] >= 11 # Dama, Valete ou Rei
      valor = valor + 10
    else
      valor = valor + mao[i]
    end
    i = i + 1
  end

  # Ajusta o valor dos Ases se necessário
  while ases > 0 && valor > 21
    valor = valor - 10
    ases = ases - 1
  end

  return valor
end
```

calculaValorMao (generic function with 1 method)

This function goes through every card in the hand and calculates the total value. First, we assume that every Ace is worth 11. If the total value goes over 21 and there are still Aces counted as 11, we convert one Ace at a time from 11 to 1 until the value fits within the limit or there are no more Aces left to convert.

Let's create helper functions to display the cards in a more friendly way:

```
function nomeCartaTexto(carta)
  nomes = ["Ás", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Dama", "Valete",
"Rei"]
```

```

        return nomes[carta]
    end

function exhibeMao(mao, jogador)
    print("$jogador possui as cartas: ")
    i = 1
    tamanhoMao = length(mao)
    while i <= tamanhoMao
        print(nomeCartaTexto(mao[i]), " ")
        i = i + 1
    end
    println()
    println("Valor da mão: $(calculaValorMao(mao))")
    println()
end

```

exibeMao (generic function with 1 method)

Now we can implement the main game logic following the rules we defined earlier.

```

function jogarBlackjack()
    baralho = criaBaralho()
    maoJogador = Int[]
    maoDealer = Int[]

    push!(maoJogador, pegaCarta(baralho))
    push!(maoDealer, pegaCarta(baralho))
    push!(maoJogador, pegaCarta(baralho))
    push!(maoDealer, pegaCarta(baralho))

    println("=== BLACKJACK ===")
    println()

    println("Dealer possui a carta: $(nomeCartaTexto(maoDealer[1]))")
    println()

    exhibeMao(maoJogador, "Você")

    valorJogador = calculaValorMao(maoJogador)
    while valorJogador < 21
        print("Deseja pedir mais uma carta? (s/n): ")
        resposta = readline()

        if resposta == "s" || resposta == "S"
            novaCarta = pegaCarta(baralho)
            push!(maoJogador, novaCarta)
            valorJogador = calculaValorMao(maoJogador)
            println("Você recebeu: $(nomeCartaTexto(novaCarta))")
            println()
            exhibeMao(maoJogador, "Você")
        else
            break
        end
    end

    if valorJogador > 21
        println("Você estourou! Perdeu o jogo.")
        return
    end

    println("Vez do dealer...")
end

```

```

println()

exibeMao(maoDealer, "Dealer")

valorDealer = calculaValorMao(maoDealer)
while valorDealer < 17
    novaCarta = pegaCarta(baralho)
    push!(maoDealer, novaCarta)
    valorDealer = calculaValorMao(maoDealer)
    println("Dealer recebeu: $(nomeCartaTexto(novaCarta))")
    println()
    exibeMao(maoDealer, "Dealer")
end

if valorDealer > 21
    println("Dealer estourou! Você ganhou!")
elseif valorJogador > valorDealer
    println("Você ganhou com $valorJogador pontos contra $valorDealer do
dealer!")
elseif valorDealer > valorJogador
    println("Dealer ganhou com $valorDealer pontos contra seus $valorJogador
pontos!")
else
    println("Empate! Ambos fizeram $valorJogador pontos.")
end
end

jogarBlackjack (generic function with 1 method)

```

The main function `jogarBlackjack()` coordinates the entire flow of the game. First, it creates a new deck and initializes empty hands. Next, it deals two cards to each player and shows the initial state, hiding one of the dealer's cards to mimic a real game.

During the player's turn, the program asks whether the player wants more cards, until the player stops or busts (goes over 21). During the dealer's turn, the rules are automatic: the dealer must keep drawing cards while the total value is below 17.

Finally, the program compares the final values and determines the winner. If both players bust, the dealer wins (the standard Blackjack rule).

To test our game, we just need to call the main function:

```
jogarBlackjack()
```

14.2 Finding the Best Strategy through Simulation

Now that we have a working game, we can use simulation to answer an interesting question: what is the best strategy for Blackjack? Instead of playing thousands of games by hand (which would be impossible), we will create several automatic strategies and simulate thousands of games to find out which one works best. We will define different strategies as functions that return the final value of the hand after applying the strategy in question.

The first strategy will be more conservative. The player keeps only the first two cards and never draws another one.

```

function estrategial(baralho)
    mao = Int[]
    push!(mao, pegaCarta(baralho))
    push!(mao, pegaCarta(baralho))
    return calculaValorMao(mao)
end

```

estrategia1 (generic function with 1 method)

For the other players, we will use more aggressive strategies, meaning the player keeps drawing cards until reaching a predetermined value, for example 21, 19, 17, 15, and 13.

```
function estrategia2(baralho, valorMaximo)
    mao = Int[]
    push!(mao, pegaCarta(baralho))
    push!(mao, pegaCarta(baralho))

    while calculaValorMao(mao) < valorMaximo
        push!(mao, pegaCarta(baralho))
    end
    return calculaValorMao(mao)
end
```

estrategia2 (generic function with 1 method)

Now that we have the strategies, we can define a match that applies the different strategies and returns each player's score. This will be useful for finding the winner.

```
function partida()
    baralho = criaBaralho()

    jogadores = zeros{Int8, 6}
    jogadores[1] = estrategia1(baralho)
    jogadores[2] = estrategia2(baralho, 21)
    jogadores[3] = estrategia2(baralho, 20)
    jogadores[4] = estrategia2(baralho, 19)
    jogadores[5] = estrategia2(baralho, 18)
    jogadores[6] = estrategia2(baralho, 17)

    return jogadores
end
```

partida (generic function with 1 method)

To find the winner, we can compare the values of all the players, checking who busted and who got closest to 21.

```
function vencedor(jogadores)
    totalJogadores = length(jogadores)
    resultado = zeros{Int8, totalJogadores}

    maximo = 0
    i = 1

    # Descobre a maior pontuação válida
    while i <= totalJogadores
        pontuacaoAtual = jogadores[i]

        if pontuacaoAtual > 21
            pontuacaoAtual = 0
        end

        if pontuacaoAtual > maximo
            maximo = pontuacaoAtual
        end

        i = i + 1
    end
```

```

i = 1

# Marca os jogadores com a maior pontuação como vencedores
while i <= totalJogadores
    pontuacaoAtual = jogadores[i]

    if pontuacaoAtual == maximo
        resultado[i] = 1
    end

    i = i + 1
end

return resultado
end

```

vencedor (generic function with 1 method)

The vencedor function returns a vector with the winners. This vector has every entry equal to zero (0), except for the winner's entry, whose value is one (1). Note that in the event of a tie, we consider the tied players all to be winners.

Finally, we can simulate thousands of matches and find out which strategy is best.

```

function melhorEstrategia()
    numeroPartidas = 100000
    contagemResultados = zeros{Int64, 6}

    i = 1
    while i <= numeroPartidas
        contagemResultados = contagemResultados + vencedor(partida())
        i = i + 1
    end

    println(contagemResultados / numeroPartidas)
end

```

melhorEstrategia()

```
[0.1292, 0.1788, 0.26986, 0.27986, 0.27338, 0.25763]
```

The simulation results show which strategy works best at Blackjack. The most conservative strategy (always stopping after 2 cards) wins only 13% of the time, while the optimal strategy, stopping at 19 points, wins 28% of the matches. Being too aggressive and drawing cards up to 21 doesn't work well either, winning only 18% of the time.

There is a balance point between being too cautious and too risky. Stopping at 19 points offers the best results, more than doubling the odds of winning compared to the conservative strategy.

15. Continuing the Modeling

```
function jogador1(cards)
    carta1 = pegarCarta(cards)
    carta2 = pegarCarta(cards)
    if carta1 == 1 || carta2 == 1
        return carta1 + carta2 + 10
    else
        return carta1 + carta2
    end
end
```

jogador1 (generic function with 1 method)

Notice that above, we used the strategy of using the Ace in the most advantageous way.

For the other players, we'll use more elaborate strategies, that is, the player keeps taking cards while they haven't reached a predetermined value, such as 21, 19, 17, 15, and 13.

Since each player can have several cards, and the sum needs to account for an Ace in the most advantageous way, we'll use a function that takes a vector of cards and calculates the sum.

```
function somaCartas(c)
    soma = 0
    temAz = false
    for i in c
        soma += i
        if c == 1
            temAz = true
        end
    end
    if soma <= 11 && temAz
        return soma + 10
    else
        return soma
    end
end
```

somaCartas (generic function with 1 method)

With the card-sum function in hand, we can model the players.

```
function jogador2(cards)
    cartas = []
    push!(cartas, pegarCarta(cards))
    push!(cartas, pegarCarta(cards))
    while somaCartas(cartas) < 21
        push!(cartas, pegarCarta(cards))
    end
    return somaCartas(cartas)
end
```

```
function jogador3(cards)
    cartas = []
    push!(cartas, pegarCarta(cards))
    push!(cartas, pegarCarta(cards))
    while somaCartas(cartas) < 19
        push!(cartas, pegarCarta(cards))
    end
    return somaCartas(cartas)
end
```

```

function jogador4(cards)
    cartas = []
    push!(cartas, pegarCarta(cards))
    push!(cartas, pegarCarta(cards))
    while somaCartas(cartas) < 17
        push!(cartas, pegarCarta(cards))
    end
    return somaCartas(cartas)
end

function jogador5(cards)
    cartas = []
    push!(cartas, pegarCarta(cards))
    push!(cartas, pegarCarta(cards))
    while somaCartas(cartas) < 15
        push!(cartas, pegarCarta(cards))
    end
    return somaCartas(cartas)
end

function jogador6(cards)
    cartas = []
    push!(cartas, pegarCarta(cards))
    push!(cartas, pegarCarta(cards))
    while somaCartas(cartas) < 13
        push!(cartas, pegarCarta(cards))
    end
    return somaCartas(cartas)
end

```

jogador6 (generic function with 1 method)

Now that we have all the players, we can model a game. To do this we create a deck and have each player follow their strategy.

```

function partida()
    cards = criaBaralho()
    jogadores = zeros{Int8, 6}
    jogadores[1] = jogador1(cards)
    jogadores[2] = jogador2(cards)
    jogadores[3] = jogador3(cards)
    jogadores[4] = jogador4(cards)
    jogadores[5] = jogador5(cards)
    jogadores[6] = jogador6(cards)
end

```

partida (generic function with 1 method)

There wasn't time to continue. It was left for the next class.

In the previous chapter, we ended up with a game, but without checking the winner, that is, the player with the highest value, less than or equal to 21. A design decision is to say that in the case of a tie, the players with the highest values win and split the prize.

```

function partida()
    cards = criaBaralho()
    jogadores = zeros{Int8, 6}
    jogadores[1] = jogador1(cards)
    jogadores[2] = jogador2(cards)
    jogadores[3] = jogador3(cards)
    jogadores[4] = jogador4(cards)

```

```

jogadores[5] = jogador5(cards)
jogadores[6] = jogador6(cards)
return jogadores
end

```

partida (generic function with 1 method)

The game returns the score of each player, so we can check in the winner routine who won.

```

function ganhador(v)
    i = 1
    maximo = 0
    while i <= length(v)
        if v[i] > 21 # se estourou é como se tivesse o menor valor
            v[i] = 0
        end
        if v[i] > maximo
            maximo = v[i] # encontra o vencedor
        end
        i = i + 1
    end
    result = zeros{Int64, length(v)}
    i = 1
    while i <= length(v)
        if v[i] == maximo
            result[i] = 1
        end
        i = i + 1
    end
    return result
end

```

ganhador (generic function with 1 method)

The winner routine returns a vector with the winners, with 1 in the position of whoever won and zero in the position of the losers.

One of the advantages of using a computer is that we can run thousands of games of 21 to find out what would be the best strategy.

```

function porcentagem()
    i = 1
    porc = zeros{Int64, 6}
    while i < 100000
        porc = porc + ganhador(partida())
        i = i + 1
    end
    println(porc)
end

```

porcentagem (generic function with 1 method)

By simulating the game 10000 times, we can find out which is the best strategy among the ones presented.

The code above ended up relatively large, and there is a lot of duplication in the code of the Players from the second one onward. One of the biggest problems with code is duplication. In the case above, we can avoid it by adding a parameter to the Player function, so that it becomes the limit to be considered in the loop. The jogador2 function then looks like this:

```

function jogador2(cards, valor)
    cartas = []

```

```

    push!(cartas, pegarCarta(cards))
    push!(cartas, pegarCarta(cards))
    while somaCartas(cartas) < valor
        push!(cartas, pegarCarta(cards))
    end
    return somaCartas(cartas)
end
jogador2 (generic function with 2 methods)

```

Since the function has a new parameter, we have to fix the game. But now we can use all the values.

```

function partida()
    cards = criaBaralho()
    jogadores = zeros{Int8, 6}
    jogadores[1] = jogador1(cards)
    jogadores[2] = jogador2(cards, 21)
    jogadores[3] = jogador2(cards, 20)
    jogadores[4] = jogador2(cards, 19)
    jogadores[5] = jogador2(cards, 18)
    jogadores[6] = jogador2(cards, 17)
    return jogadores
end

```

partida (generic function with 1 method)

Notice that there's no change to the winner function, which keeps working.

To finish, we can now have an interactive version that allows a human player to play against the computer.

```

function partidaComHumano()
    cards = criaBaralho()
    humano = []
    computador = jogador2(cards, 19)
    push!(humano, pegarCarta(cards))
    push!(humano, pegarCarta(cards))
    println("O humano tem ", humano, " e soma ", somaCartas(humano))
    println("O humano quer mais cartas (S/N)?")
    resp = readline()
    while resp == "S" || resp == "s"
        push!(humano, pegarCarta(cards))
        println("O computador tem ", computador, " e soma ",
somaCartas(computador))
        println("O humano tem ", humano, " e soma ", somaCartas(humano))
        println("O humano quer mais cartas (S/N)?")
        resp = readline()
    end
    println("O computador tem ", computador, " e soma ", somaCartas(computador))
    if somaCartas(computador) <= 21 && somaCartas(humano) <= 21
        if somaCartas(computador) > somaCartas(humano)
            println("Humano Perdeu")
        elseif somaCartas(computador) == somaCartas(humano)
            println("Empate")
        else
            println("Humano ganhou")
        end
    elseif somaCartas(computador) > 21 && somaCartas(humano) > 21
        println("os dois perderam")
    elseif somaCartas(computador) > 21
        println("Humano ganhou")
    else

```

```
        println("Computador ganhou")
    end
end

partidaComHumano (generic function with 1 method)
```


16. Good Practices

This chapter presents three good programming practices. They come from a whole field dedicated to software development called Software Engineering.

16.1 Using contracts

Whenever possible, code should be modular, that is, split into files and/or functions. Each function should make clear what its parameters are and what it returns. This can be done using types.

```
function fatorial(n::Int64)::Int64
    if n < 2
        return 1
    else
        return n * fatorial(n - 1)
    end
end
```

fatorial (generic function with 1 method)

With this, it becomes clear what the function receives and returns, and if a different type than expected is passed in, we get an immediate error.

16.2 Good practice 1: Use types

16.3 Automated tests

To prevent errors, or bugs, from creeping in, an effective approach is to write code that checks whether the code works. If this is done automatically, we have automated tests.

```
using Test
function testaFat()
    @test fatorial(3) == 6
    @test fatorial(5) == 120
    @test fatorial(1) == 1
    @test fatorial(0) == 1
    @test fatorial(4) == 24
end
```

testaFat (generic function with 1 method)

16.4 Good practice 2: Write tests whenever possible

16.5 Write code for humans, not for computers

Even though computers are capable of reading code that isn't always well formatted, it's quite hard for humans to read code that doesn't follow a standard. So here are some important tips:

- Use indentation. This makes blocks clear and makes it easy to identify loops, if blocks, and function bodies.
- Choose variable and function names carefully. This makes the code much easier for others to read.
- Whenever you spot a chance to improve the code, do it. It's even better if you have automated tests, so you can check that the improvement didn't break the code.

16.6 Good practice 3: Write code for others to read

16.7 Applying good practices

We'll now solve the following problem, applying the practices above. Given a vector of real numbers, determine which numbers appear in the vector and the number of times each of them occurs in it.

Analyzing the problem, we see that the input is a vector of real numbers, which may contain repetitions. To determine which numbers are in the vector, we can use another vector as output. Both the input and the output vectors should be of type Float64. In addition, for the vector that gives the count of each number we need a vector of integers. With that, we already have the function's signature.

```
function conthist(v::Vector{Float64}, el::Vector{Float64}, qtd::Vector{Int64})  
end
```

conthist (generic function with 1 method)

With this signature in hand, we can already write the tests.

```
function verifica(v::Vector{Float64}, elementos::Vector{Float64},  
    quant::Vector{Int64})  
    el = Float64[]  
    quan = Int64[]  
    conthist(v, el, quan)  
    if el == elementos && quan == quant  
        return true  
    else  
        return false  
    end  
end
```

```
function testalista()  
    @test verifica([1.3, 1.2, 0.0, 1.3], [1.3, 1.2, 0.0], [2, 1, 1])  
    @test verifica([1.0, 1.0, 1.0, 1.0], [1.0], [4])  
    @test verifica([8.3], [8.3], [1])  
    @test verifica([3.14, 2.78, 2.78], [3.14, 2.78], [1, 2])  
end
```

testalista (generic function with 1 method)

Finally, we can write the code. The idea behind the solution is simple: we'll go through the input vector. For each element, there are two possibilities. If it hasn't appeared before, we add the number to the output vector and record one occurrence. If it has already appeared, we just increase the occurrence count.

```
function conthist(v::Vector{Float64}, el::Vector{Float64}, qtd::Vector{Int64})  
    for a in v  
        if a in el  
            i = 1  
            while el[i] != a  
                i += 1  
            end  
            qtd[i] += 1  
        else  
            push!(el, a)  
            push!(qtd, 1)  
        end  
    end  
end
```

contHist (generic function with 1 method)

17. Going Beyond One Dimension (Matrices)

So far we have worked with structures that have more than one dimension, but without looking closely at their type. In this lesson we will try to understand the differences between them and how this can be used to our advantage.

Let's start with lists:

```
v = [1, 2, 3]
typeof(v)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

The type returned is: Vector{Int64} (alias for Array{Int64, 1}). In this case this means that v is a vector of integers, or a one-dimensional array. Likewise

```
v = zeros{Int64, 3}
typeof(v)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

But vectors can be more flexible, as in the example below:

```
v = [1, 2.0, "três"]
typeof(v)
```

```
Vector{Any} (alias for Array{Any, 1})
```

In this case the vector's type is no longer integers and becomes "Any", that is, Vector{Any} (alias for Array{Any, 1}).

Furthermore, imagine the following situation:

```
a = [1, 2, 3]
push!(v, a)
typeof(v)
```

```
Vector{Any} (alias for Array{Any, 1})
```

In this case, the vector remains of type Any, but in the fourth position we now have a vector with three integers. This shows us that vector structures can be quite flexible. However, this flexibility with different types usually comes at some cost, generally in performance.

On the other hand, structures can also have more than one dimension. These are called matrices, and they can be created with the zeros function that we already used above.

```
m = zeros{Int64, 3, 2}
typeof(m)
```

```
Matrix{Int64} (alias for Array{Int64, 2})
```

Above we created a two-dimensional matrix with three rows and two columns. Its elements can be accessed just like in a vector, but now with two indices.

```
m[1, 2] = 10
10
```

```
function imprime(m::Array{Int64,2})
    println(m)
end
```

```
imprime (generic function with 1 method)
```

```
function imprime(m::Vector{Vector{Int64}})
    println(m[1])
end
```

```

        println(m[2])
    end

imprime (generic function with 2 methods)
function imprime(m::Vector{Vector{Int64}})
    for i in m
        println(i)
    end
end

imprime (generic function with 2 methods)
function imprime(m::Vector{Vector{Int64}})
    for i in m
        for j in m[i]
            println(j, " ")
        end
    end
end

imprime (generic function with 2 methods)
function imprime(m::Vector{Vector{Int64}})
    for i in m
        print("|")
        for j in i
            print(j, " ")
        end
        println("|")
    end
end

imprime (generic function with 2 methods)
function imprimeMatriz(m::Matrix{Int64})
    println(m)
end

imprimeMatriz (generic function with 1 method)
function imprimeMatriz(m::Matrix{Int64})
    i = 1
    while i < size(m)[1]
        println(m[1])
        i += 1
    end
end

imprimeMatriz (generic function with 1 method)
function imprimeMatriz(m::Matrix{Int64})
    i = 1
    while i < size(m)[1]
        j = 1
        while j < size(m)[2]
            print(m[i, j], " ")
            j += 1
        end
        println()
        i += 1
    end
end

imprimeMatriz (generic function with 1 method)

```

```

function preencheMatriz(m::Matrix{Int64})
    i = 1
    while i <= length(m)
        m[i] = rand{Int} % 10
        i += 1
    end
end

preencheMatriz (generic function with 1 method)

function criaIdentidade(tam::Int64)
    m = zeros{Int64, tam, tam}
    i = 1
    while i <= tam
        m[i, i] = 1
    end
    return m
end

criaIdentidade (generic function with 1 method)

Direct operations on matrices such as +, - and *

```


18. String Exercises

In this lesson, we'll explore functions that manipulate strings and write tests to check their correctness. For some functions, we'll notice that there are several ways to get the same result.

18.1 Concatenating letters

The first function, `concatena`, concatenates the first two and the last two letters of a string.

```
function concatena(s::String)::String
    if length(s) < 2
        return "Erro: tamanho da string menor do que 2"
    end
    resposta = s[1:2]*s[end-1:end]
    return resposta
end
```

`concatena` (generic function with 1 method)

Here we use `s[1:2]` to get the first two letters of `s`, which is a more concise way to access more than one index of an object. Alternatively, we could access these two indices separately with the command `s[1]*s[2]`.

To check whether the function is working correctly, we can use the following test:

`using Test`

```
function testeConcatena()
    @test concatena("Ola Bom Dia") == "Olia"
    @test concatena("oi") == "oioi"
    @test concatena("tre") == "trre"
    @test concatena("a") == "Erro: tamanho da string menor do que 2"
    @test concatena("a123") == "a123"
end
```

`testeConcatena` (generic function with 1 method)

18.2 String reversal

We need to create a function that reverses a string, returning the characters in reverse order.

```
function inverta(s::String)::String
    # Inicializamos uma string vazia
    inversa=""

    # Intervalo de length(s) até 1, a passos de -1
    for i in length(s):-1:1
        # Concatena cada caractere na ordem inversa
        inversa*=s[i]
    end

    return inversa
end
```

`inverta` (generic function with 1 method)

To get the result we want, we build a for loop that iterates from the last index of the string, `length(s)`, down to the first, using a step of `-1`. At each iteration, we concatenate the current character, `s[i]`, onto the string `inversa`, so the characters end up in reverse order.

Now we can write a test function to check that our `inverta` function works correctly.

```
using Test

function testeInverte()
    @test inverte("123") == "321"
    @test inverte("x") == "x"
    @test inverte("SOS") == "SOS"
    @test inverte("tres") == "sert"
end

testeInverte (generic function with 1 method)
```

18.3 The reverse function

Julia already provides a function called `reverse`, which can be used to reverse both vectors and strings. For example:

```
reversa = reverse("exemplo")

"olpmexe"
```

In this example, the `reverse` function receives only the object to be reversed as a parameter, but in the case of vectors, we can also specify exactly which range we want reversed.

```
vetor = [1, 2, 3, 4, 5]
reversa = reverse(vetor, 2, 4)

5-element Vector{Int64}:
 1
 4
 3
 2
 5
```

18.4 Modifying a string

The third function, `modifica`, alters a string that ends with “ing” by adding “ly”, or otherwise adds “ing”.

```
function modifica(s::String)::String
    if length(s) < 3
        return "Erro: tamanho da string menor do que 3"
    end

    if s[end-2:end] == "ing"
        s = s*"ly"
    else
        s = s*"ing"
    end

    return s
end

modifica (generic function with 1 method)
```

In this example, we manually check the last three characters of the string `s`. However, Julia offers a more practical and readable function called `endswith`, which we can use to simplify this check.

```
function modifica(s::String)::String
    if length(s) < 3
        return "Erro: tamanho da string menor do que 3"
    end
```

```

    if endswith(s, "ing")
        s = s*"ly"
    else
        s = s*"ing"
    end

    return s
end

modifica (generic function with 1 method)

```

Let's now write the test that checks that the functions above work correctly.

```

using Test
function testaModifica()
    @test modifica("doing") == "doingly"
    @test modifica("sing") == "singly"
    @test modifica("run") == "runing"
    @test modifica("talk") == "talking"
end

testaModifica (generic function with 1 method)

```

18.5 Rearranging letters

The second function, `rearranja`, receives a string and returns a string that contains the lowercase letters first, followed by the uppercase letters.

We can check whether a letter is uppercase or lowercase using the ASCII table, which encodes characters as integers. In the table, uppercase letters fall in the range 65 to 90, and lowercase letters in the range 97 to 122.

To learn more about the ASCII table you can visit [this page](#).

```

function rearranja(s::String)::String
    maiusculos=""
    minusculos=""

    for i in 1:length(s)
        if Int(s[i]) >= 65 && Int(s[i]) <= 90
            maiusculos = maiusculos*s[i]
        elseif Int(s[i]) >= 97 && Int(s[i]) <= 122
            minusculos = minusculos*s[i]
        end
    end

    return minusculos*maiusculos
end

rearranja (generic function with 1 method)

```

A more readable approach is to use the `islowercase` and `isuppercase` functions, which check whether a letter is lowercase or uppercase, respectively.

```

function rearranja(s::String)::String
    maiusculos=""
    minusculos=""

    for i in 1:length(s)
        if isuppercase(s[i])
            maiusculos = maiusculos*s[i]
        elseif islowercase(s[i])

```

```

        minusculos = minusculos*s[i]
    end
end

return minusculos*maiusculos

end

rearranja (generic function with 1 method)

```

We can then write the test for our functions.

```

using Test

function testaRearranja()
    @test rearranja1("PaRaLe10") == "aaelPRL0"
    @test rearranja1("ELEfantE") == "fantELEE"
    @test rearranja1("0lá") == "l0"
    @test rearranja1("13La2") == "aL"
    @test rearranja2("PaRaLe10") == "aaelPRL0"
    @test rearranja2("ELEfantE") == "fantELEE"
    @test rearranja2("0lá") == "láo"
    @test rearranja2("13La2") == "aL"
end

testaRearranja (generic function with 1 method)

```

18.6 Finding the longest word

Our last function should receive a list of words and return the longest one, along with its length.

```

function maior_palavra(vetor::Vector{String})
    # Inicialmente, a maior palavra que encontramos é uma string vazia
    maior_palavra = ""
    maior_tamanho = 0

    for palavra in vetor
        # Verifica se a palavra atual é maior que a maior encontrada até agora
        if length(palavra) > maior_tamanho
            maior_palavra = palavra
            maior_tamanho = length(palavra)
        end
    end

    return maior_palavra, maior_tamanho
end

maior_palavra (generic function with 1 method)

```

Although this code looks correct, it doesn't handle the case where there is more than one word with the greatest length. For example:

```

vetor = ["boa", "bem", "oi"]
maior_palavra(vetor)

("boa", 3)

```

In this case, only the word "boa" is returned, even though "bem" has the same length. To fix this, we need to change the variable that stores the longest word so it can hold more than one word. We'll use a vector of strings for that.

```

function maiores_palavras(vetor::Vector{String})
    maiores_palavras = String[]
    maior_tamanho = 0

    for palavra in vetor
        # Se a palavra é maior do que o maior tamanho salvo,
        # então todas as palavras que estão no vetor maior_palavra são menores do
        # que a palavra atual
        if length(palavra) > maior_tamanho
            # Limpa o vetor e salva a palavra atual
            maiores_palavras = String[]
            push!(maiores_palavras, palavra)
            maior_tamanho = length(palavra)

            # Se é igual ao tamanho salvo, então é do mesmo tamanho que as palavras já
            # salvas no vetor maiores_palavras,
            # apenas damos push na palavra atual
            elseif length(palavra) == maior_tamanho
                push!(maiores_palavras, palavra)
            end
        end
    end

    return maiores_palavras, maior_tamanho
end

```

maiores_palavras (generic function with 1 method)

Now we can write tests for this last function.

using Test

```

function testeMaioresPalavras()
    vetor1 = ["gato", "elefante", "cachorro"]
    @test maiores_palavras(vetor1) == (["elefante", "cachorro"], 8)

    vetor2 = ["a", "ab", "abc"]
    @test maiores_palavras(vetor2) == (["abc"], 3)

    vetor3 = ["bem", "boa", "bom", "oi"]
    @test maiores_palavras(vetor3) == (["bem", "boa", "bom"], 3)

    vetor4 = ["", " ", "teste"]
    @test maiores_palavras(vetor4) == (["teste"], 5)

    vetor5 = String[]
    @test maiores_palavras(vetor5) == ([], 0)

    vetor6 = ["a", "ab", "abc", "xyz", "xy"]
    @test maiores_palavras(vetor6) == (["abc", "xyz"], 3)
end

```

testeMaioresPalavras (generic function with 1 method)

18.7 Returning multiple values

As we saw in the previous exercise, Julia allows a function to return multiple values. This lets you send back more than one result from a function call, making the code more concise and easier to understand. This is especially useful for things like mathematical operations, decompositions, or data processing, which often need more than one result.

To return multiple values in Julia, you can simply separate them with commas. Here's a simple example:

```
function troca(a,b)
    aux = a
    a = b
    b = aux

    return a, b
end
```

troca (generic function with 1 method)

When calling this function, you can capture the multiple returned values in separate variables:

```
a, b = troca(1, 10)
(10, 1)
```


19. Appendix: Answers to the question “What is a computer?”

The answers were collected through the site <https://www.bli.do/MAC115>. Here are the answers:

- “It’s a machine that uses certain patterns to process information. It’s an evolution of the theoretical Turing machine and can be considered an extension of our intelligence.”
- “Computer: a machine capable of carrying out binary processes of creation and analysis, modeled either by the device itself (AI) or by a programmer. A product of steps.”
- “A computer is a machine capable of identifying codes based on binary values using logical language.”
- “It’s a technological instrument that gathers millions of pieces of data shared by people all over the world, making it possible to work in a technical way, creating new data and ways of reaching answers.”
- “A device that lives in the outlet.”
- “It’s equipment that ‘runs’ an installed program that receives data, processes it, and outputs a result, if everything goes well...”
- “Decoder.”
- “Its purpose is to be a tool based on algorithms and decision logic that, once programmed and set up, can enable problem solving through the collection of data and information, and then the generation of systems.”
- “A computer is an electronic device characterized by its logical processing capacity. They are built with semiconductors and can have several different parts, such as a power supply (or battery), motherboard, processor, graphics card, memory, and so on.”
- “A robot that will control humans one day.”
- “A device that does arithmetic.”
- “A computer is a device that computes, that does arithmetic, using a CPU (Central Processing Unit) to perform calculations using the binary system. 0s and 1s, which are used in logic gates made of silicon, a semiconductor that allows calculations using these two digits to be carried out.”
- “A machine.”
- “It’s a thing that has tools that help carry out different tasks.”
- “It’s a 2.0 phone.”
- “A computer is a machine that obeys commands.”
- “A calculator.”
- “Turing machine.”
- “A computer is, in a very simplified way, an electronic intelligence made up of a shell (hardware) that is completed by the software, built to respond to commands.”
- “The evolution of the calculator.”