

Programming in Julia & Jupyter Notebooks

ROB 102: Introduction to AI & Programming

Lab Session 7

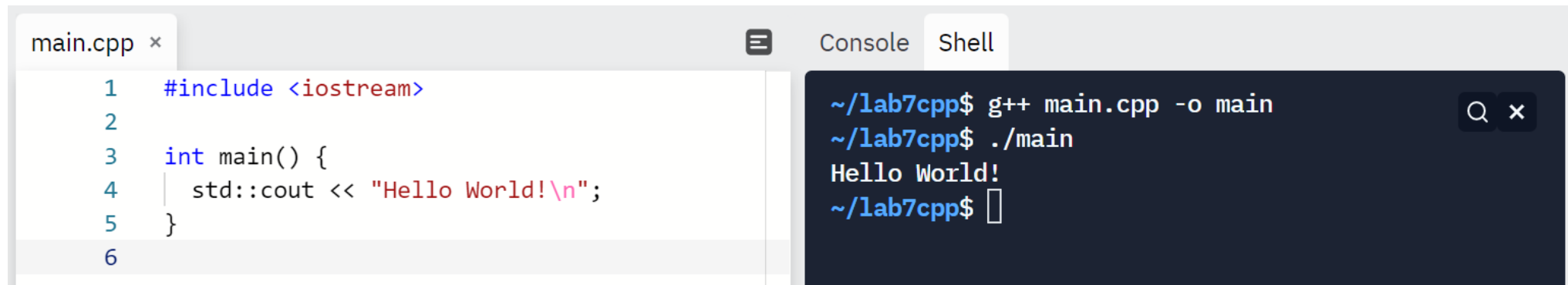
2021/11/19

Today

1. Coding in Julia
2. Using a Jupyter Notebook

Running code in Julia vs. C++

In C++, we need to compile our code into an executable and then run it.



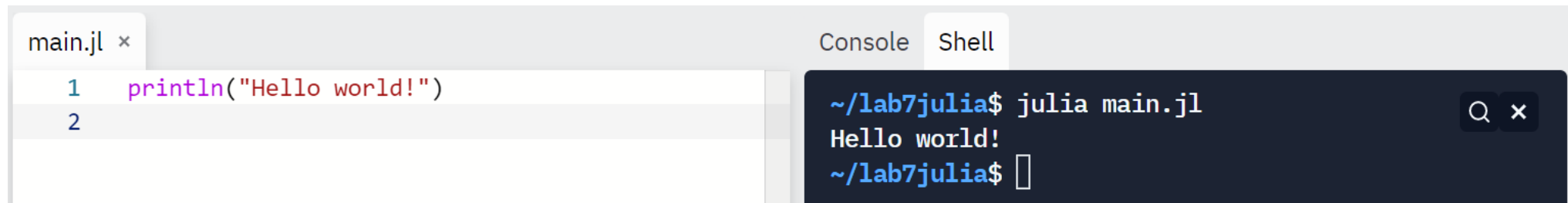
The screenshot shows a code editor with a file named `main.cpp`. The code is as follows:

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello World!\n";
5 }
6
```

To the right of the editor is a console window with the following output:

```
~/lab7cpp$ g++ main.cpp -o main
~/lab7cpp$ ./main
Hello World!
~/lab7cpp$
```

In Julia, code is executed line by line without a compiler. Julia scripts do not need a main function.



The screenshot shows a code editor with a file named `main.jl`. The code is as follows:

```
1 println("Hello world!")
2
```

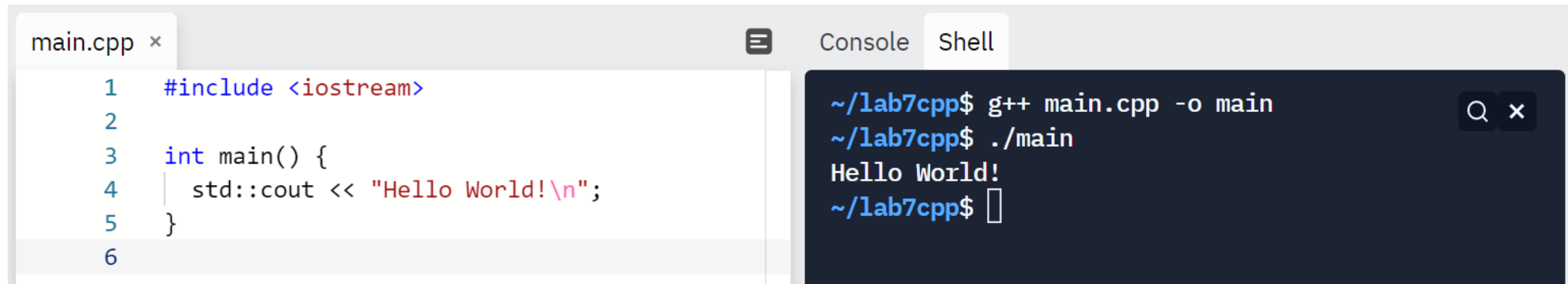
To the right of the editor is a console window with the following output:

```
~/lab7julia$ julia main.jl
Hello world!
~/lab7julia$
```

Statements in Julia do not need a semi-colon at the end of them.

Printing in Julia vs. C++

In C++, we print using `std::cout`.



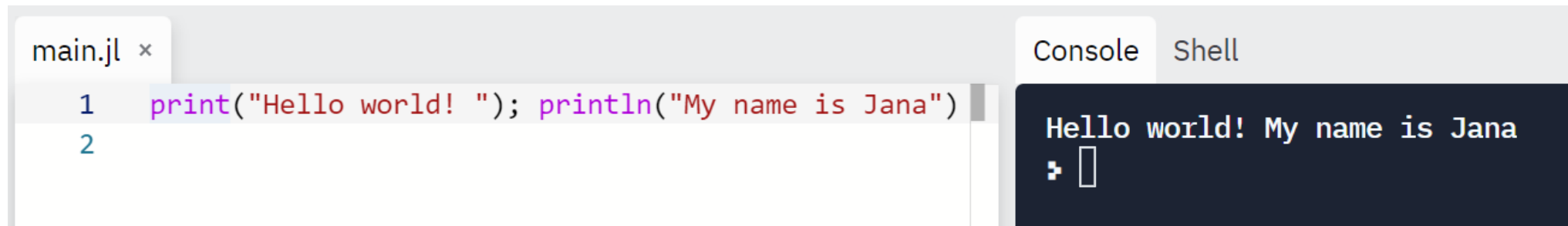
The screenshot shows a C++ IDE with a file named `main.cpp`. The code in the editor is as follows:

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello World!\n";
5  }
6
```

To the right of the editor is a terminal window with tabs for 'Console' and 'Shell'. The 'Shell' tab is active, showing the following commands and output:

```
~/lab7cpp$ g++ main.cpp -o main
~/lab7cpp$ ./main
Hello World!
~/lab7cpp$
```

In Julia, the `println("...")` function prints the content and then skips a line. `print("...")` prints without skipping a line.



The screenshot shows a Julia IDE with a file named `main.jl`. The code in the editor is as follows:

```
1  print("Hello world! "); println("My name is Jana")
2
```

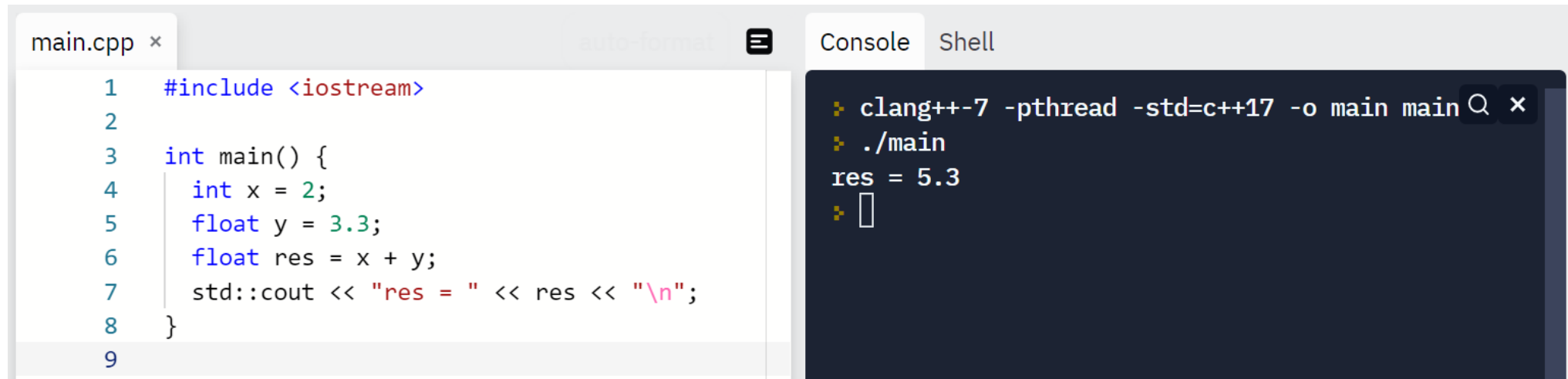
To the right of the editor is a terminal window with tabs for 'Console' and 'Shell'. The 'Shell' tab is active, showing the following output:

```
Hello world! My name is Jana
>
```

In Julia, you can include a semi-colon to indicate the end of a line.

Variables & Types

In C++, variables need to be declared with their types.

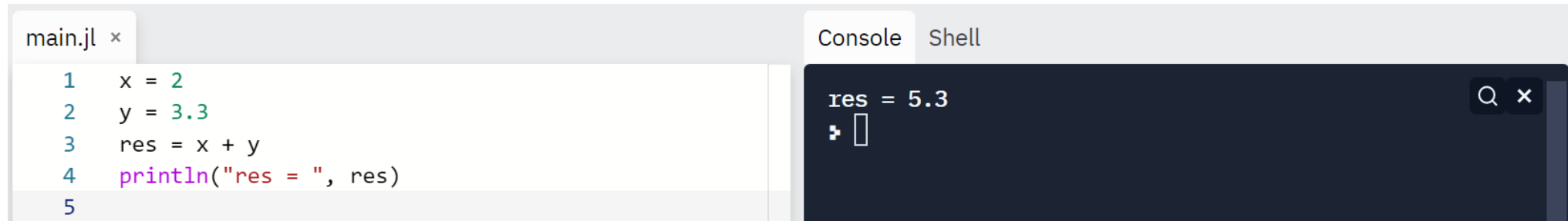


```
main.cpp x auto-format
1  #include <iostream>
2
3  int main() {
4      int x = 2;
5      float y = 3.3;
6      float res = x + y;
7      std::cout << "res = " << res << "\n";
8  }
9
```

```
Console Shell
❯ clang++-7 -pthread -std=c++17 -o main main
❯ ./main
res = 5.3
❯
```

Variables & Types

In Julia, variables are declared without any types. Julia figures out what the type is for you.



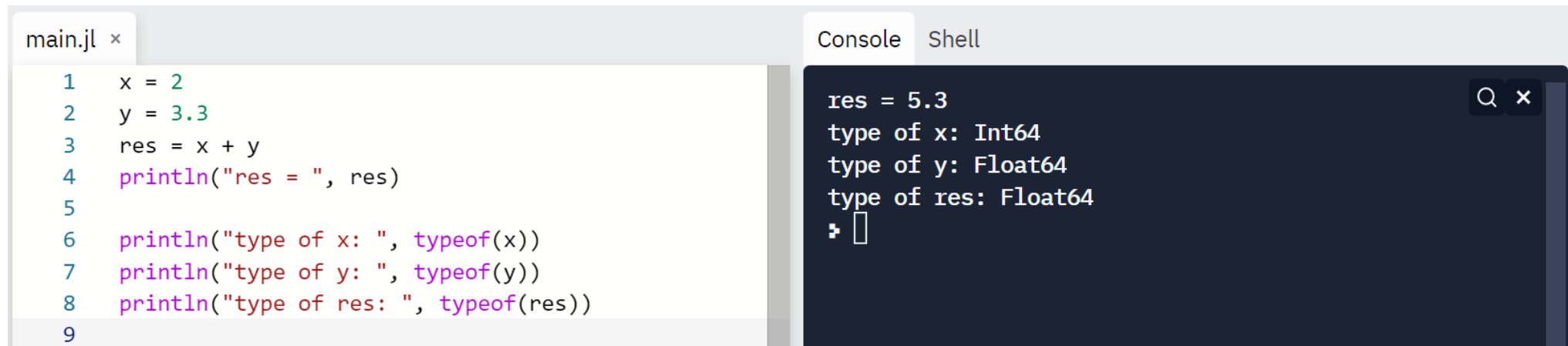
The screenshot shows a Julia REPL window with a file named `main.jl`. The code in the editor is:

```
1 x = 2
2 y = 3.3
3 res = x + y
4 println("res = ", res)
5
```

The console output shows:

```
res = 5.3
└─▶
```

You can find out the type of a variable using the `typeof(...)` function.



The screenshot shows a Julia REPL window with a file named `main.jl`. The code in the editor is:

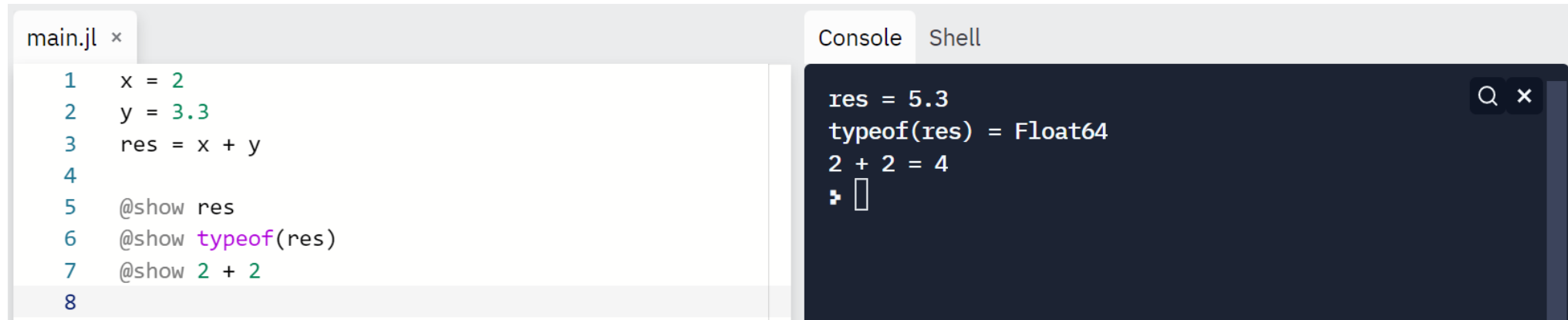
```
1 x = 2
2 y = 3.3
3 res = x + y
4 println("res = ", res)
5
6 println("type of x: ", typeof(x))
7 println("type of y: ", typeof(y))
8 println("type of res: ", typeof(res))
9
```

The console output shows:

```
res = 5.3
type of x: Int64
type of y: Float64
type of res: Float64
└─▶
```

Printing Revisited

A useful function for showing a line and the result of the line (instead of `print(...)` or `println(...)`) is `show`:



The screenshot shows a Julia REPL interface with two panes. The left pane, titled 'main.jl x', contains the following code:

```
1 x = 2
2 y = 3.3
3 res = x + y
4
5 @show res
6 @show typeof(res)
7 @show 2 + 2
8
```

The right pane, titled 'Console' and 'Shell', shows the output of the code:

```
res = 5.3
typeof(res) = Float64
2 + 2 = 4
┌─┐
```

The output demonstrates that the `@show` macro displays the expression being evaluated along with its result and type.

Operators

The standard math operators (+, -, *, /, %) are the same as in C++, except that dividing two integers gives a float. Julia also has an operator, ^, which is the same as `pow()` in C++.



The screenshot shows a Julia REPL interface. On the left, a file named 'main.jl' is open, displaying the following code:

```
1 # This is a comment!  
2 x = 5  
3 y = 2  
4 @show x / y  
5 @show x^y  
6
```

On the right, the 'Console' tab is active, showing the output of the code:

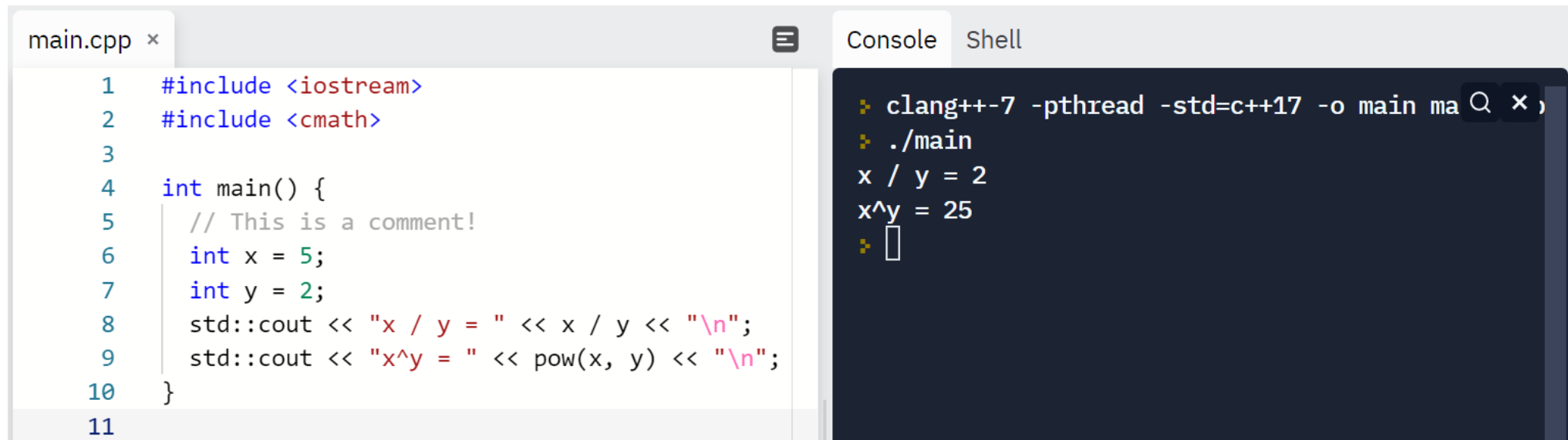
```
x / y = 2.5  
x ^ y = 25  
└─┘
```

The output shows that the division of 5 by 2 results in a float (2.5), and the power operation (5 to the power of 2) results in 25. A search icon and a close icon are visible in the top right corner of the console panel.

Julia code.

Operators

The standard math operators (+, -, *, /, %) are the same as in C++, except that dividing two integers gives a float. Julia also has an operator, ^, which is the same as pow() in C++.



The screenshot shows a C++ IDE with two panels. The left panel displays the source code in `main.cpp`, and the right panel shows the console output.

```
main.cpp x
1  #include <iostream>
2  #include <cmath>
3
4  int main() {
5      // This is a comment!
6      int x = 5;
7      int y = 2;
8      std::cout << "x / y = " << x / y << "\n";
9      std::cout << "x^y = " << pow(x, y) << "\n";
10 }
11
```

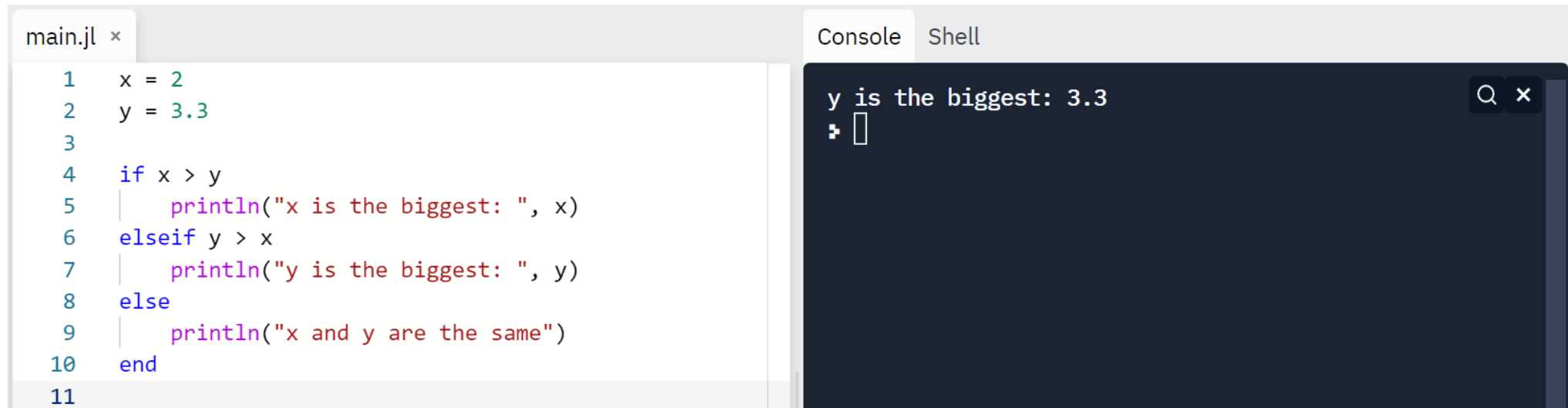
The console output shows the execution of the program:

```
clang++-7 -pthread -std=c++17 -o main ma
./main
x / y = 2
x^y = 25
```

C++ code.

Control flow: If Statements

In Julia, if statements don't need brackets around the condition or curly brackets around the code.



```
main.jl x
1  x = 2
2  y = 3.3
3
4  if x > y
5      println("x is the biggest: ", x)
6  elseif y > x
7      println("y is the biggest: ", y)
8  else
9      println("x and y are the same")
10 end
11
```

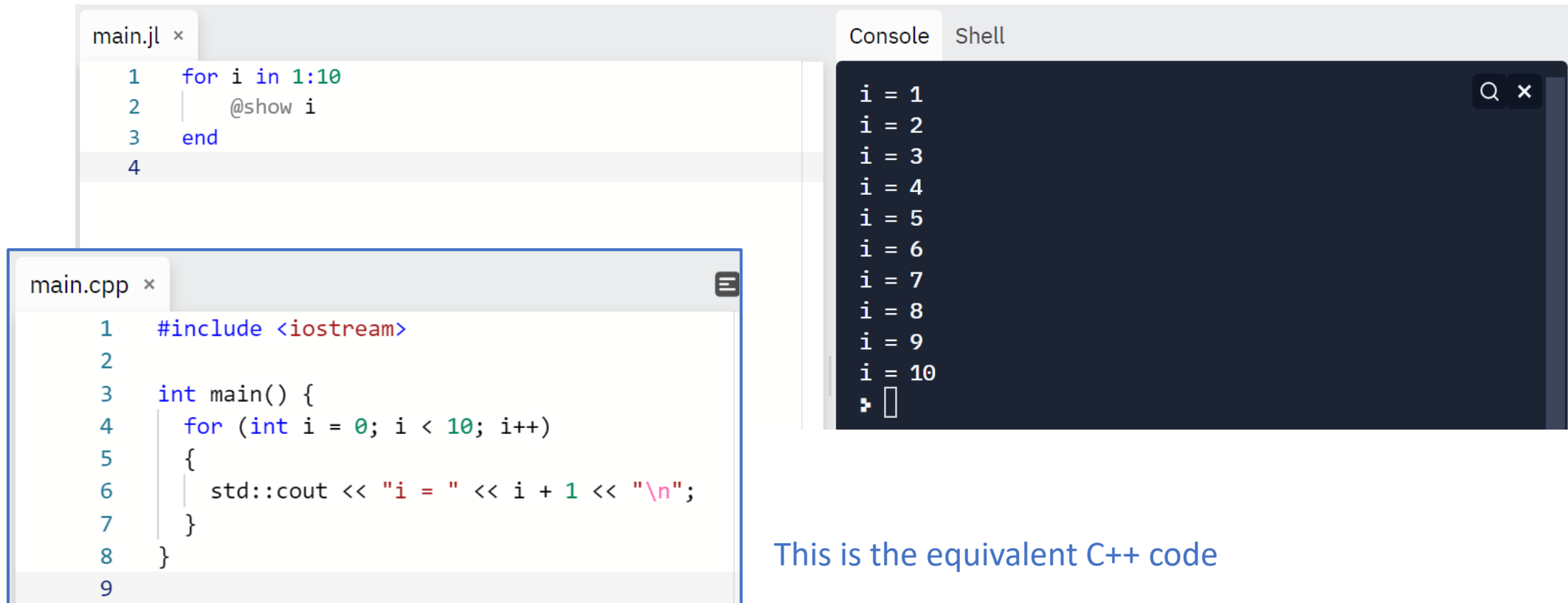
Console Shell

```
y is the biggest: 3.3
> 
```

If statements end with the end keyword. We write elseif instead of else if in C++.

Control flow: Loops

The syntax for a for loop in Julia looks like this:



The image shows a code editor with two files: `main.jl` and `main.cpp`. The `main.jl` file contains a Julia for loop that iterates from 1 to 10 and prints each value. The `main.cpp` file contains the equivalent C++ code, which includes `<iostream>`, defines a `main` function, and uses a for loop to iterate from 0 to 9, printing `i + 1` each time. To the right of the code editor is a console window showing the output of the Julia code, which is the numbers 1 through 10, each on a new line.

```
main.jl x
1  for i in 1:10
2      @show i
3  end
4

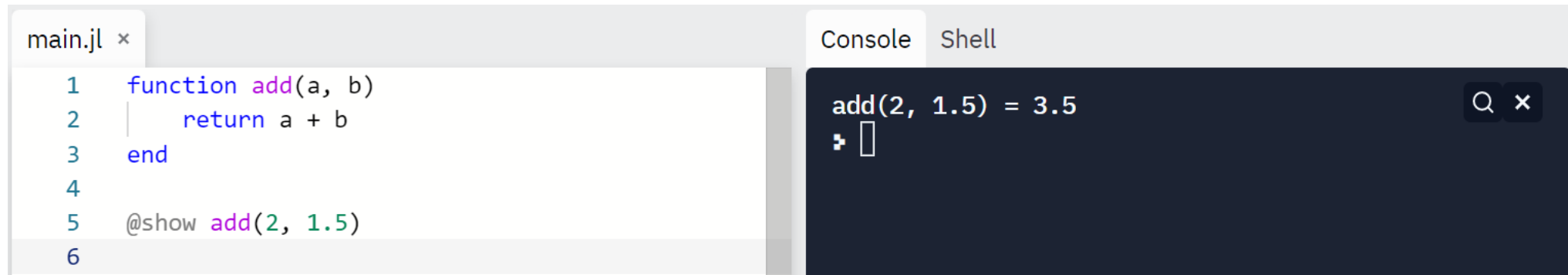
main.cpp x
1  #include <iostream>
2
3  int main() {
4      for (int i = 0; i < 10; i++)
5      {
6          std::cout << "i = " << i + 1 << "\n";
7      }
8  }
9

Console  Shell
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
❏
```

This is the equivalent C++ code

Functions

In Julia, functions are declared with the `function` keyword. The `end` keyword ends a function.



The screenshot shows a Julia REPL interface. On the left, a code editor window titled 'main.jl' contains the following code:

```
1 function add(a, b)
2     return a + b
3 end
4
5 @show add(2, 1.5)
6
```

On the right, the 'Console' tab is active, displaying the output of the function call:

```
add(2, 1.5) = 3.5
julia>
```

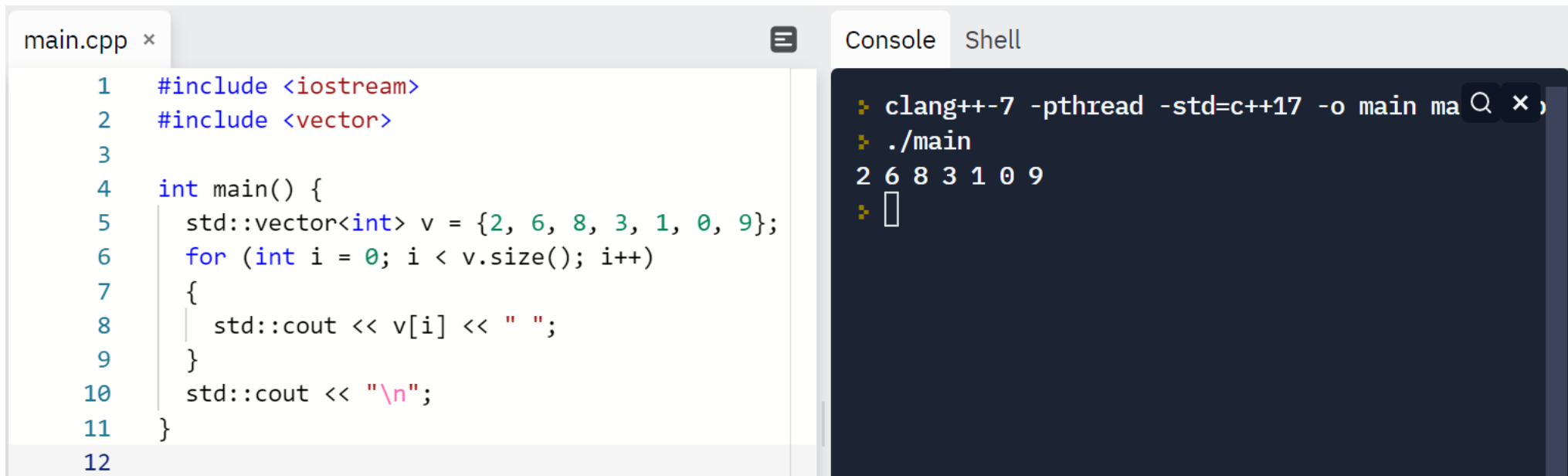
Functions do not need return types, and function parameters do not need types.

Vectors & Matrices

Julia is especially good at dealing with vectors and matrices. A vector can be created like this:

$$v = [1, 2, 3, 4]$$

In C++, we iterated through each index of the vector using its size, like this:



The screenshot shows a C++ IDE with a file named `main.cpp` and a console window. The code in `main.cpp` includes `<iostream>` and `<vector>`, and defines a `main` function that creates a vector `v` with the values `{2, 6, 8, 3, 1, 0, 9}`. It then iterates through the vector using a `for` loop from `0` to `v.size() - 1`, printing each element followed by a space. The console output shows the sequence of numbers `2 6 8 3 1 0 9` followed by a newline character.

```
main.cpp x
1  #include <iostream>
2  #include <vector>
3
4  int main() {
5      std::vector<int> v = {2, 6, 8, 3, 1, 0, 9};
6      for (int i = 0; i < v.size(); i++)
7      {
8          std::cout << v[i] << " ";
9      }
10     std::cout << "\n";
11 }
12
```

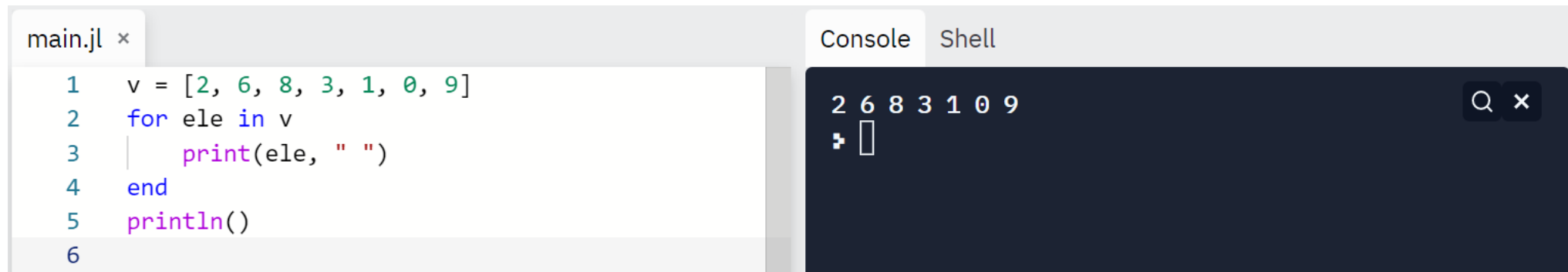
```
Console Shell
❯ clang++-7 -pthread -std=c++17 -o main ma
❯ ./main
2 6 8 3 1 0 9
❯
```

Vectors & Matrices

Julia is especially good at dealing with vectors and matrices. A vector can be created like this:

$$v = [1, 2, 3, 4]$$

In Julia, we can iterate through elements in a vector directly:



The screenshot shows a Julia REPL interface with two panels: 'main.jl' and 'Console'. The 'main.jl' panel contains the following code:

```
1 v = [2, 6, 8, 3, 1, 0, 9]
2 for ele in v
3     print(ele, " ")
4 end
5 println()
6
```

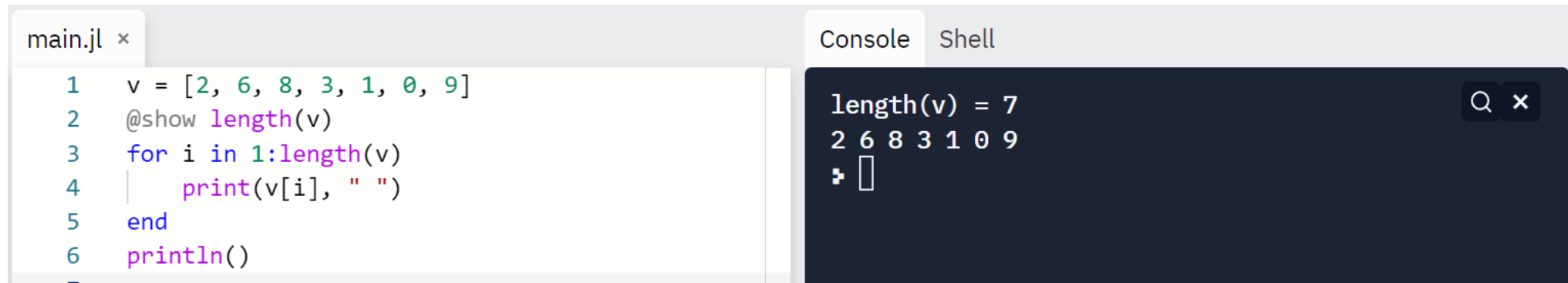
The 'Console' panel shows the output of the code, which is the elements of the vector printed in a single line: '2 6 8 3 1 0 9'. Below the output, there is a prompt character 'julia>' followed by an empty box, indicating that the REPL is ready for the next command.

Vectors & Matrices

Julia is especially good at dealing with vectors and matrices. A vector can be created like this:

$$v = [1, 2, 3, 4]$$

We can iterate by index as well, using `length(v)` to get the length.



The screenshot shows a Julia REPL window with a file named `main.jl`. The code in the editor is as follows:

```
1 v = [2, 6, 8, 3, 1, 0, 9]
2 @show length(v)
3 for i in 1:length(v)
4     print(v[i], " ")
5 end
6 println()
```

The console output on the right shows the results of the code execution:

```
length(v) = 7
2 6 8 3 1 0 9
␣
```

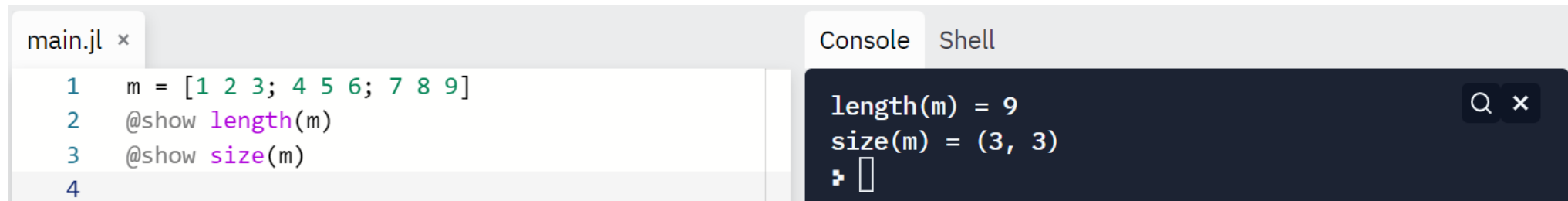
In Julia, vectors are indexed starting at 1. `v[1]` is the first element, not `v[0]`.

Vectors & Matrices

Creating a matrix is similar to creating a vector. Semi-colons separate rows, and spaces separate elements in a row:

```
m = [1 2 3; 4 5 6; 7 8 9]
```

The length of a matrix is the total number of elements in it. The size of the matrix is a tuple containing the number of rows and columns.



The screenshot shows a Julia REPL window with a file named 'main.jl'. The code in the editor is as follows:

```
1 m = [1 2 3; 4 5 6; 7 8 9]
2 @show length(m)
3 @show size(m)
4
```

The right side of the window shows the 'Console' output:

```
length(m) = 9
size(m) = (3, 3)
┌ []
```

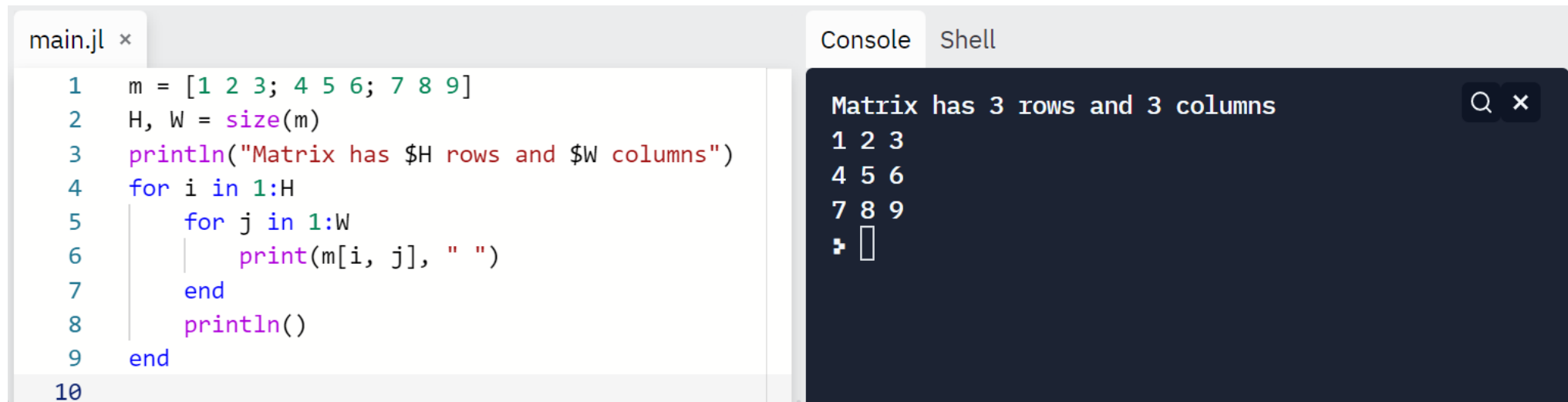
A *tuple* is like a vector, but it doesn't allow its values to be modified.

Vectors & Matrices

Creating a matrix is similar to creating a vector. Semi-colons separate rows, and spaces separate elements in a row:

$$m = [1 \ 2 \ 3; 4 \ 5 \ 6; 7 \ 8 \ 9]$$

Indexing into a matrix can be done by providing the row and column index of a matrix, separated by commas:



The screenshot shows a Julia REPL window with two tabs: 'main.jl' and 'Console'. The 'main.jl' tab is active, displaying the following code:

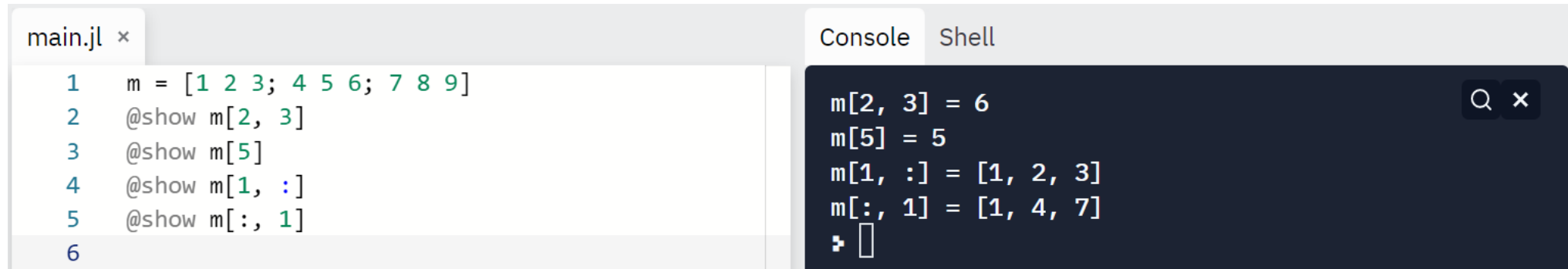
```
1 m = [1 2 3; 4 5 6; 7 8 9]
2 H, W = size(m)
3 println("Matrix has $H rows and $W columns")
4 for i in 1:H
5     for j in 1:W
6         print(m[i, j], " ")
7     end
8     println()
9 end
10
```

The 'Console' tab is also active, showing the output of the code:

```
Matrix has 3 rows and 3 columns
1 2 3
4 5 6
7 8 9
julia>
```

Vectors & Matrices

- Indexing into a matrix can be done by providing the row and column index of a matrix, separated by commas.
- Indexing into a matrix with just one value treats the matrix as a flattened vector.
- Colons let us pick all the values in an axis. `m[1, :]` is all the values in row 1. `m[:, 1]` is all the values in column 1.

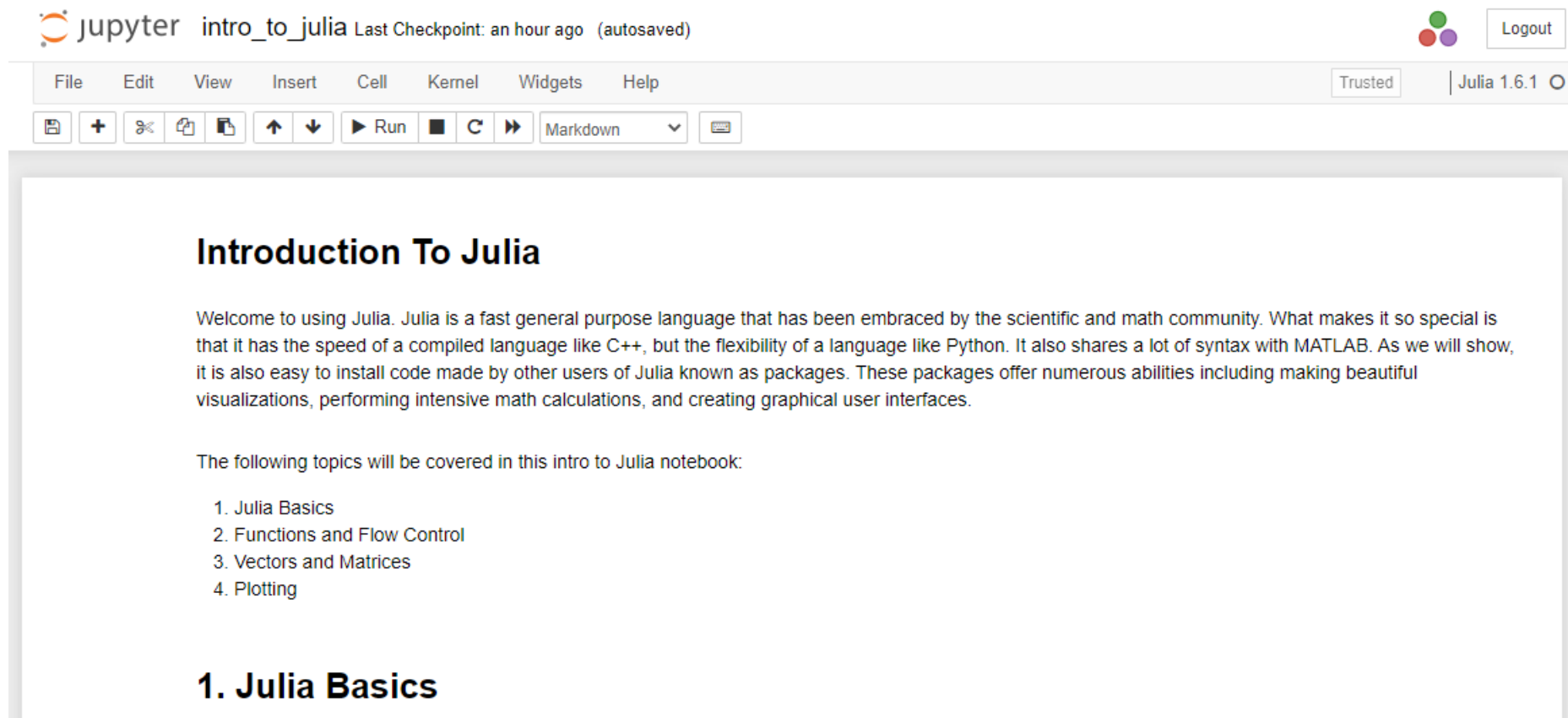


The screenshot shows a Julia REPL interface with a code editor on the left and a console on the right. The code editor, titled 'main.jl', contains six lines of code: 1. `m = [1 2 3; 4 5 6; 7 8 9]`, 2. `@show m[2, 3]`, 3. `@show m[5]`, 4. `@show m[1, :]`, 5. `@show m[:, 1]`, and 6. (empty line). The console, titled 'Console', shows the output of these commands: `m[2, 3] = 6`, `m[5] = 5`, `m[1, :] = [1, 2, 3]`, `m[:, 1] = [1, 4, 7]`, and a prompt `julia>` with an empty array `[]` below it. A search icon and a close icon are visible in the top right of the console.

```
main.jl ×  
1 m = [1 2 3; 4 5 6; 7 8 9]  
2 @show m[2, 3]  
3 @show m[5]  
4 @show m[1, :]  
5 @show m[:, 1]  
6  
Console Shell  
m[2, 3] = 6  
m[5] = 5  
m[1, :] = [1, 2, 3]  
m[:, 1] = [1, 4, 7]  
julia>   
[]
```

Jupyter Notebooks

A Jupyter is a collection of code and text cells which allows for fast execution and visualization of code.



The screenshot displays the Jupyter Notebook web interface. At the top, the header shows the Jupyter logo, the notebook name 'intro_to_julia', and the status 'Last Checkpoint: an hour ago (autosaved)'. On the right, there are three colored dots (green, red, purple) and a 'Logout' button. Below the header is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. To the right of the menu bar is a 'Trusted' status indicator and the text 'Julia 1.6.1'. Below the menu bar is a toolbar with icons for saving, adding, deleting, and running cells, as well as a 'Markdown' dropdown menu. The main content area shows the notebook's title 'Introduction To Julia' and a paragraph of introductory text about Julia. Below the text is a list of topics to be covered in the notebook. The first section, '1. Julia Basics', is highlighted.

jupyter intro_to_julia Last Checkpoint: an hour ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted | Julia 1.6.1

Save Add Delete Run Cell Kernel Widgets Help

Introduction To Julia

Welcome to using Julia. Julia is a fast general purpose language that has been embraced by the scientific and math community. What makes it so special is that it has the speed of a compiled language like C++, but the flexibility of a language like Python. It also shares a lot of syntax with MATLAB. As we will show, it is also easy to install code made by other users of Julia known as packages. These packages offer numerous abilities including making beautiful visualizations, performing intensive math calculations, and creating graphical user interfaces.

The following topics will be covered in this intro to Julia notebook:

1. Julia Basics
2. Functions and Flow Control
3. Vectors and Matrices
4. Plotting

1. Julia Basics

TODO:

1. Complete the setup instructions to install Julia and the necessary packages.
2. Accept the “Intro to Julia” assignment.
 - The assignment links are in the Google Doc linked on Slack.
3. Go through the Intro to Julia assignment and complete the exercises (optional but encouraged).