For my son

“Special Forces for Science and Knowledge”

-Arthur L.

Rust Programming for Beginners Simplified Version ©

Index

Index

Symbols used in this book

Symbols used in this book

Mind Map

Concept

Theory

Code

Question

Rust Compilation

- Rust is a systems programming language that aims for safety, performance, and concurrency. To make a simple Rust project, you have to write the Rust code, compile it, and then run the compiled binary. We'll use the basic "Hello, World" program to walk through the compilation process.

1. Write or create a Rust source code file:

- First, create a new file called `main.rs` and write the code inside it.

2. Compile the Rust source code:

To compile the Rust code, open a terminal (command prompt) and navigate to the directory where your `main.rs` file is located. Then, run the following command:
rustc main.rs
`rustc` is Rust's compiler - it compiles your source code into machine code, which can then be executed directly by your computer. The compiled binary will have the same name as your source file, but without the `.rs` extension, so for our example, it will result in a file named `main` (or `main.exe` on Windows).

Now that the code is compiled, you can execute the binary generated in the previous step. Run the following command:

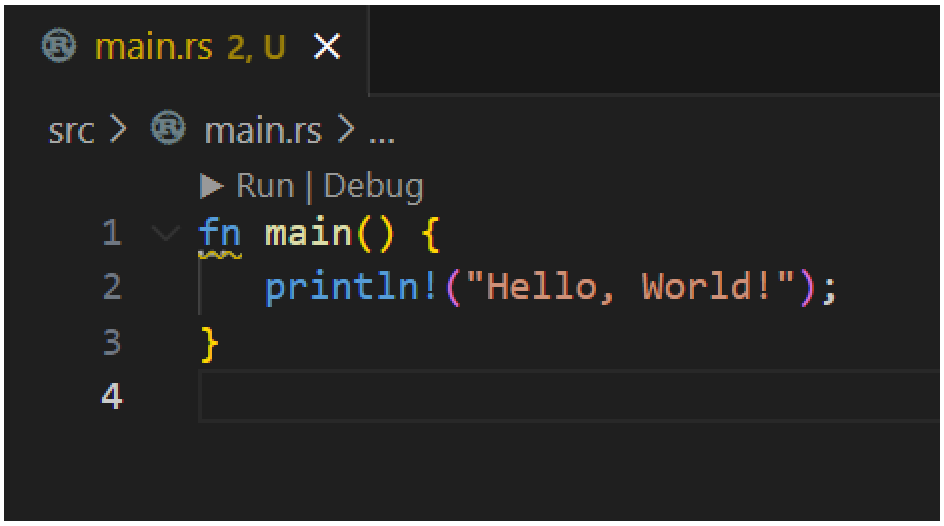
On Linux and macOS:

```
./main
```

On Windows:

```
main.exe
```

File name: main.rs

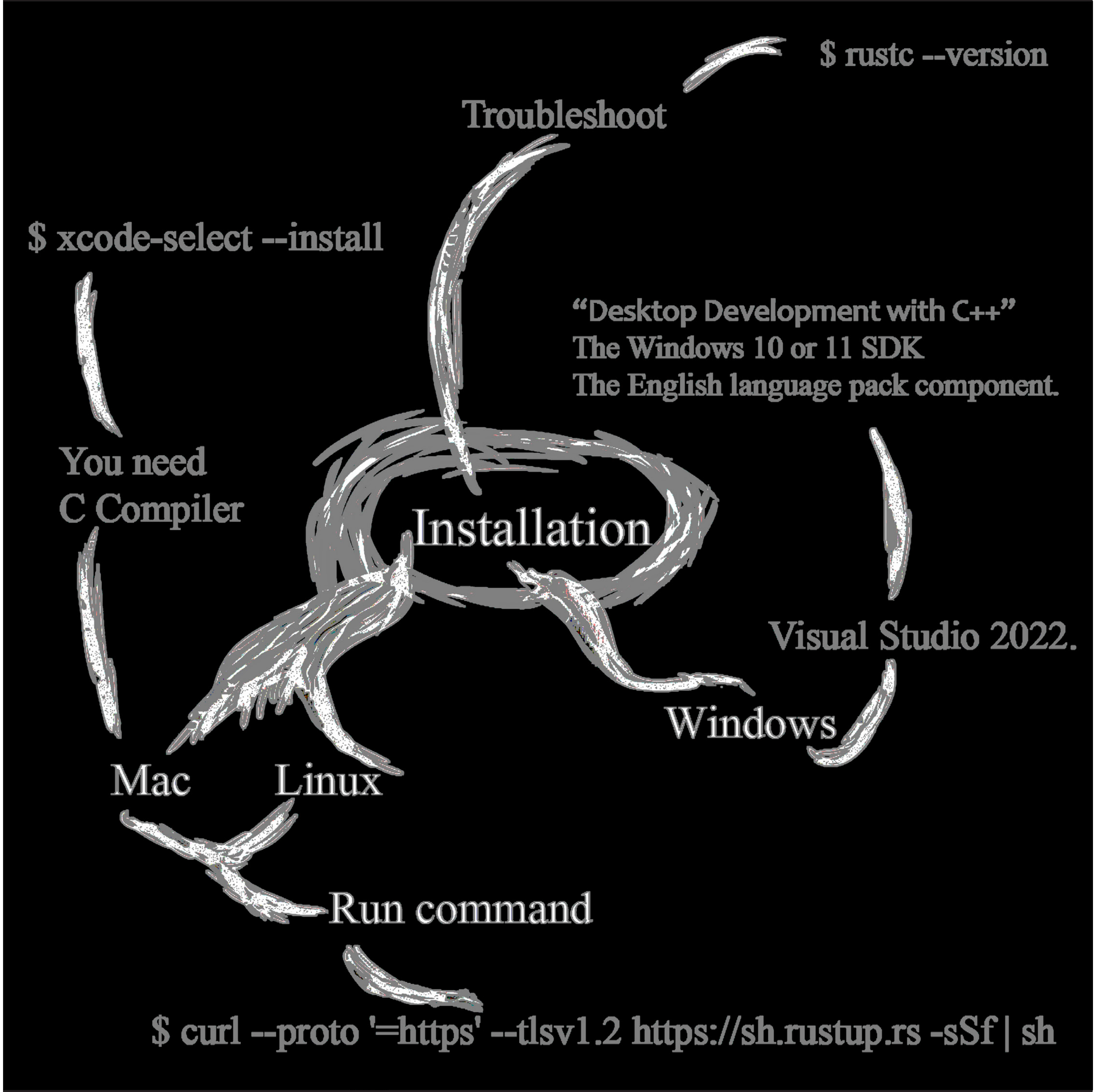


```rust
fn main() {
    println!("Hello, World!");
}
```

You should see the output "Hello, World!" printed in your terminal. Congratulations! You have successfully written, compiled, and executed a Rust program.

In summary, the compilation process in Rust involves creating a source code file (e.g., `main.rs`), using the `rustc` command to compile it, and

Rust Installation



```
$ rustc --version
```

```
$ xcode-select --install
```

“Desktop Development with C++”
The Windows 10 or 11 SDK
The English language pack component.

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Rust Installation

FOR INSTALLING RUST

Mac /Linux

Run the command on terminal:

```
$ curl --proto ‘=https’ --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Mac

Run the command on terminal (get a C compiler):

```
$ xcode-select --install
```

Windows

You’ll need Visual Studio 2022

- “Desktop Development with C++”
- The Windows 10 or 11 SDK
- The English language pack component

Troubleshoot

Get Rust installed version on terminal:

```
$ rustc --version
```

Update

```
$ rustup update
```

Uninstall

```
$ rustup self uninstall
```

Rust installation

- The rustup tool is used for installing and managing Rust.

Rustup is a tool that helps you to install and manage different versions of the Rust Programming Language from official release channels. With Rustup, it’s easy to switch between stable, beta, and nightly compilers, and keep them up to date.

Cargo

Cargo is the default package manager and build tool for the Rust programming language. It helps developers manage their project's dependencies, compile source code, run tests, and distribute their compiled applications easily. Essentially, Cargo streamlines the entire development process in Rust, allowing programmers to focus more on coding.

Cargo

Building Cargo consists of several commands, including:

1. `**cargo build**`: This command compiles your Rust project and generates an executable binary in the "target/debug" directory. It also downloads and compiles all dependencies specified in the "Cargo.toml" file.

2. `**cargo run**`: This command not only builds the project but also immediately runs the compiled binary. It is particularly useful during development because it simplifies the process of testing your code.

`**cargo check**` is another useful command that analyzes your source code and determines if it has any syntax or type errors without actually building it. This works much faster than the `cargo build` command because it does not create an executable binary, which can save developers a significant amount of time during the development process.

Building for release is an essential step in the deployment of a Rust application. To optimize your code for better performance, use the command `**cargo build --release**`. This prepares your project with optimizations, creating a more efficient binary in the "***target/release***" directory that is ideal for distribution.

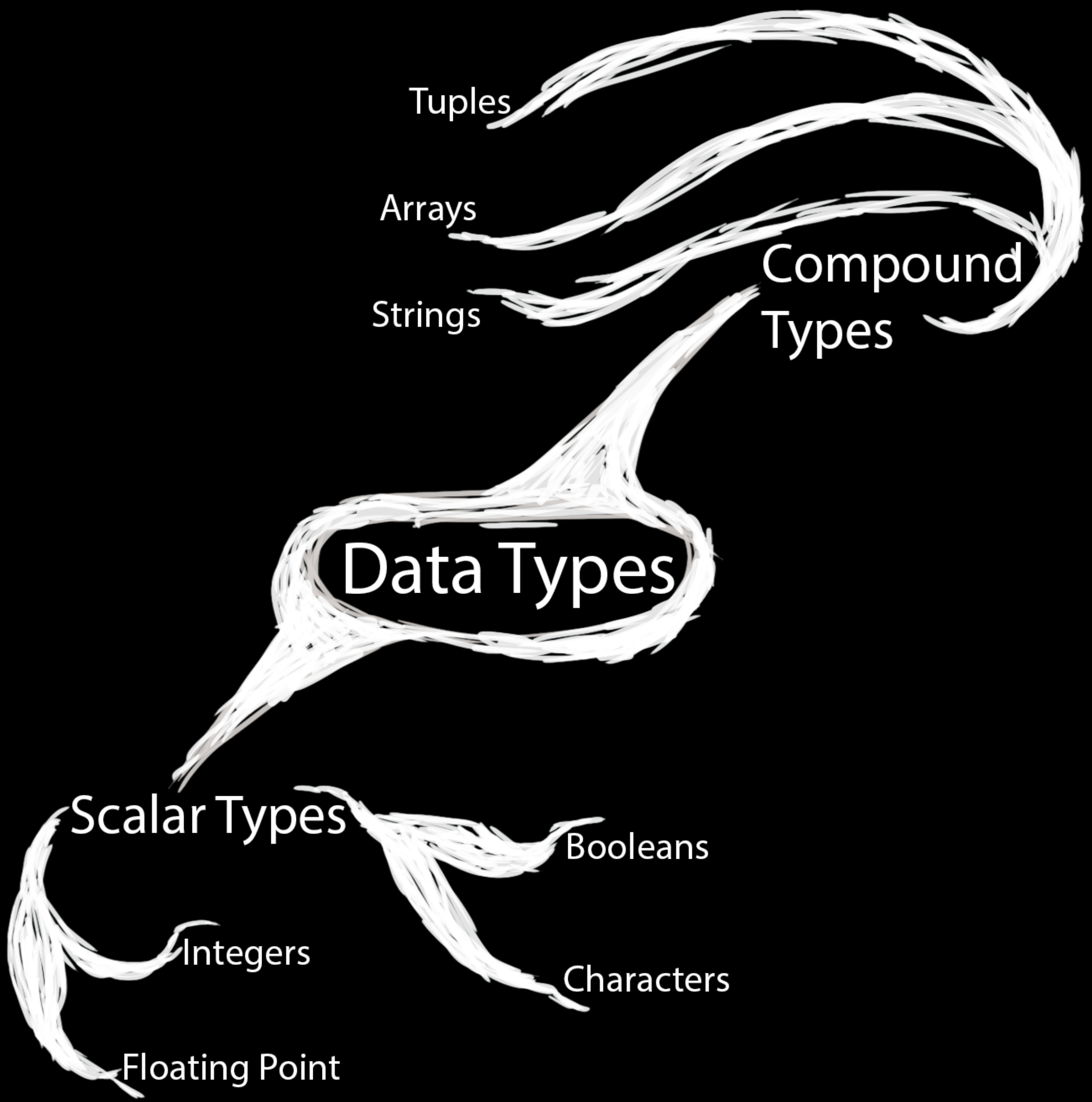
Data Types

Scalar Types:

- Integers (whole numbers)
- Floating-point numbers (decimals)
- Booleans (true/false)
- Characters (individual letters/symbols)

Primitive Compound Types

- Tuples
- Arrays


Tuples
Arrays
Strings
Compound Types
Data Types
Scalar Types
Booleans
Integers
Characters
Floating Point

Data Types

Rust has several data types: scalars and compounds. Scalar types include integers, floating-point numbers, booleans, and characters. Compound types structure multiple values: tuples group together values of different types, and arrays hold multiple values of the same type. Strings, a special type, store sequences of characters. All these types help programmers model real-world problems in their code.

Scalar Types

Scalar types in Rust consist of four main categories:

- integers (whole numbers)
- floating-point numbers (decimals)
- Booleans (true/false)
- Characters (individual letters/symbols). These basic data types help you manipulate and represent values and are essential for programming in Rust.

Bit Concept

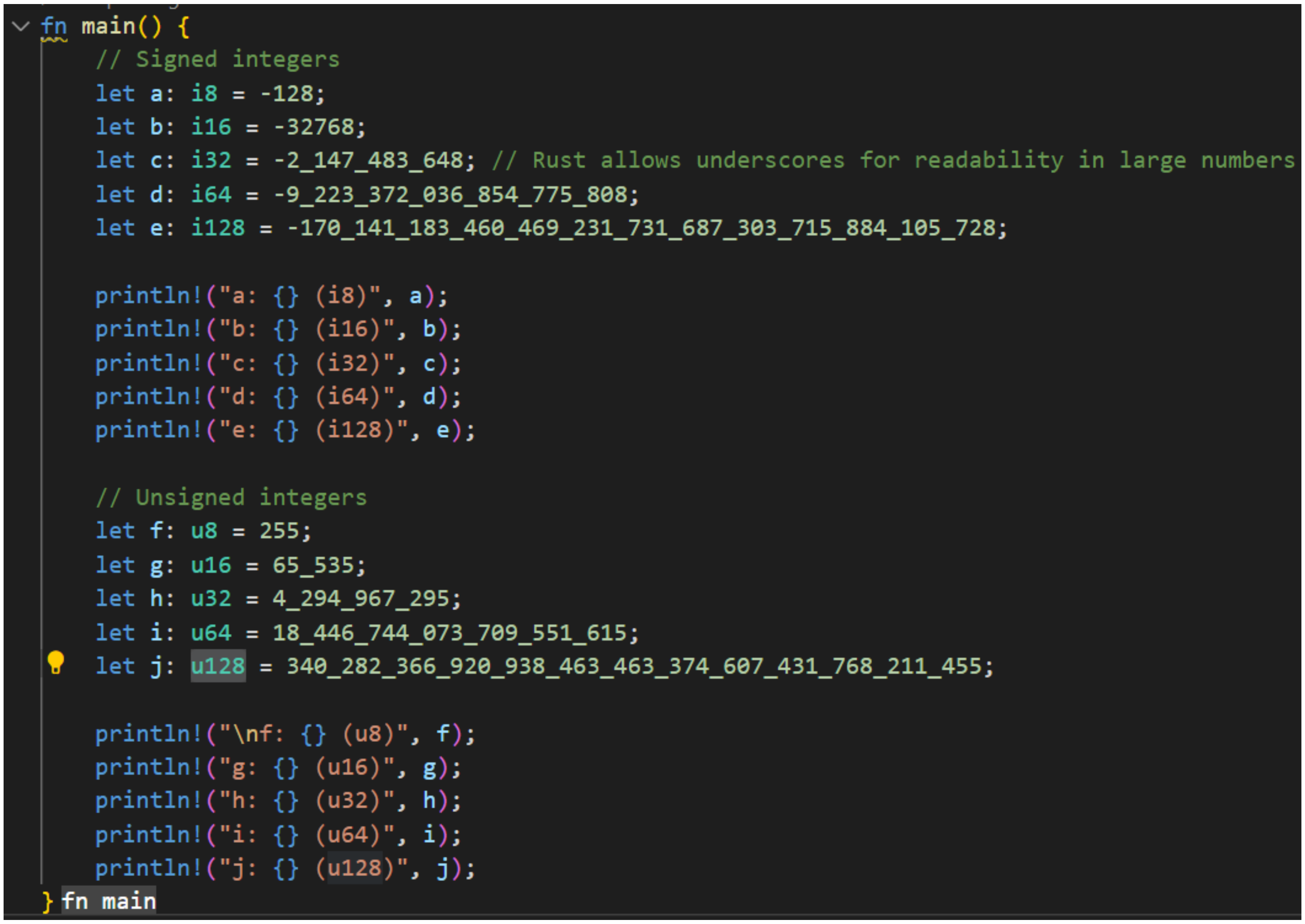
A bit is the fundamental unit of information in computing and digital communications representing a binary state with two possible values typically “1” or “0”. It can also symbolize true/false or on/off. A bit’s relation to its physical state depends on the specific device or program. Bits form larger units like bit strings bit vectors or bit arrays. Eight bits create a byte though byte size can vary. One bit is equivalent to the information entropy of an equally probable binary variable also called a shannon. The symbol for a bit is either “bit” or lowercase “b”.

Integer Types

Rust scalar types include integer types which represent numbers without fractional parts. Integers can be signed (negative or positive) or unsigned (only positive). There are different sizes such as: 8-bit, 16-bit, 32-bit, 64-bit and 128-bit. For example i32 is a signed 32-bit integer while u64 is an unsigned 64-bit integer. The isize and usize types depend on the computer's architecture (32 or 64 bits). Integer literals can be written in: decimal, hex, octal, binary or byte forms.
If unsure about which integer type to use Rust's default i32 is a good starting point. Use isize or usize for indexing collections.

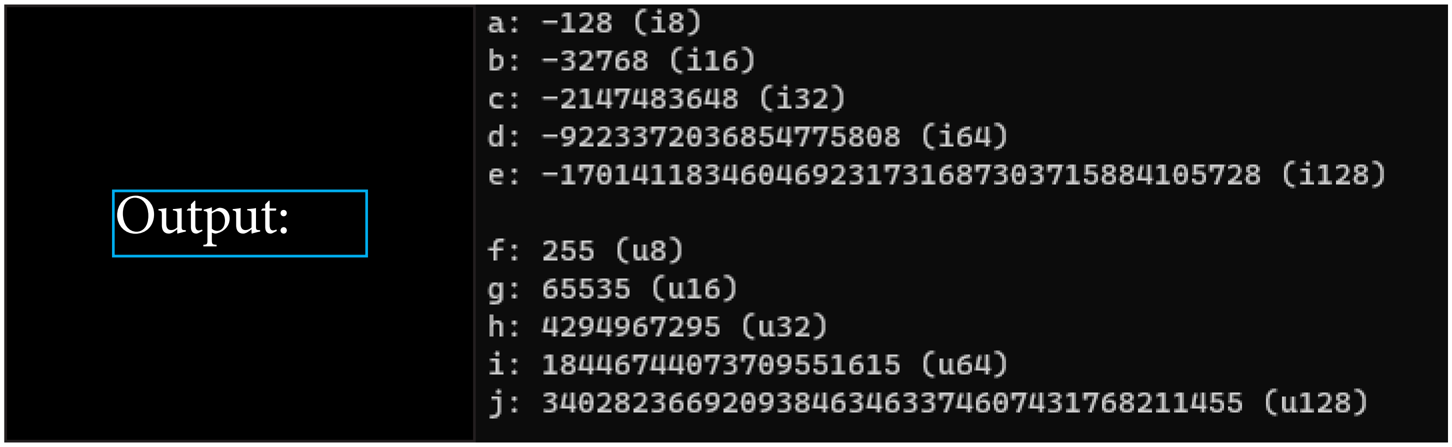
Integer Types

File name: main.rs



```rust
fn main() {
    // Signed integers
    let a: i8 = -128;
    let b: i16 = -32768;
    let c: i32 = -2_147_483_648; // Rust allows underscores for readability in large numbers
    let d: i64 = -9_223_372_036_854_775_808;
    let e: i128 = -170_141_183_460_469_231_731_687_303_715_884_105_728;

    println!("a: {} (i8)", a);
    println!("b: {} (i16)", b);
    println!("c: {} (i32)", c);
    println!("d: {} (i64)", d);
    println!("e: {} (i128)", e);

    // Unsigned integers
    let f: u8 = 255;
    let g: u16 = 65_535;
    let h: u32 = 4_294_967_295;
    let i: u64 = 18_446_744_073_709_551_615;
    let j: u128 = 340_282_366_920_938_463_463_374_607_431_768_211_455;

    println!("\nf: {} (u8)", f);
    println!("g: {} (u16)", g);
    println!("h: {} (u32)", h);
    println!("i: {} (u64)", i);
    println!("j: {} (u128)", j);
} fn main
```

$ cargo run



Output:

```
a: -128 (i8)
b: -32768 (i16)
c: -2147483648 (i32)
d: -9223372036854775808 (i64)
e: -170141183460469231731687303715884105728 (i128)

f: 255 (u8)
g: 65535 (u16)
h: 4294967295 (u32)
i: 18446744073709551615 (u64)
j: 340282366920938463463374607431768211455 (u128)
```

Large integers

How to work with Really Big Numbers for: Science, Research, Space, Physics, Astronomy; using Rust?

Large integers

In simple terms, Rust has some basic building blocks (called primitives) which have a fixed size in terms of bits. A bit is a tiny piece of computer memory that can store either 0 or 1. The number of bits determine how many different values you can store using that type of building block.

For example, Rust has a type called `i64`, which uses 64 bits, and it can store positive and negative whole numbers. The largest number it can store is 2^63-1, which is a huge number but not as huge as 99e100 (which is a 1 followed by 100 zeroes).

Similarly, Rust has other types like `i128` for even larger whole numbers, and `u128` for even larger non-negative whole numbers. However, none of these can store numbers as large as 99e100.

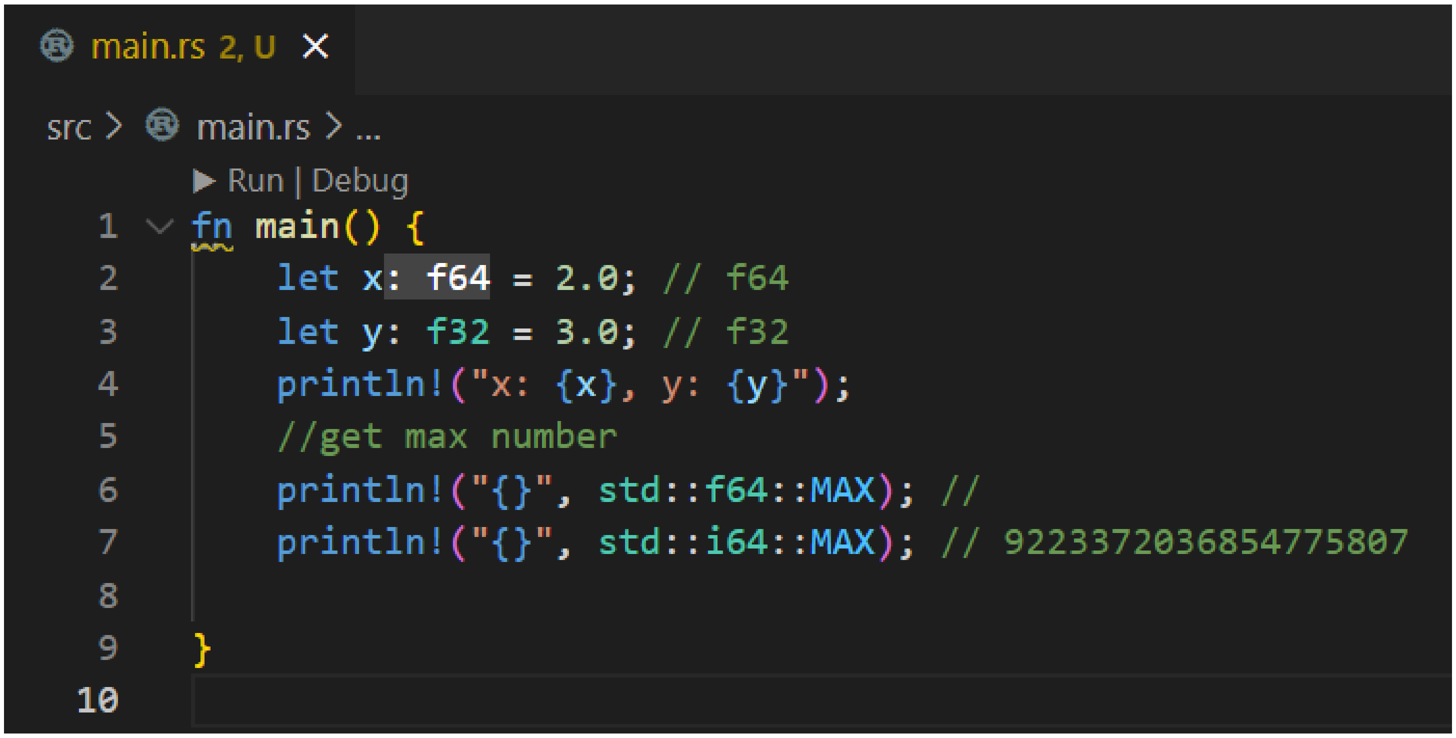
If you need to work with really big numbers, you'll need to use some special types that can grow as needed. These types are not included by default in Rust and you'll either need to create them yourself, or use a package that someone else already made, like the `num` crate.

Floating-point numbers

Floating-point numbers are values with decimal points, and Rust supports two types: f32 (32 bits) and f64 (64 bits). By default, Rust uses f64, offering better precision and similar speed to f32 on modern CPUs. Both types are signed and abide by the IEEE-754 standard. For example, declaring variables ‘x’ as f64 and ‘y’ as f32:

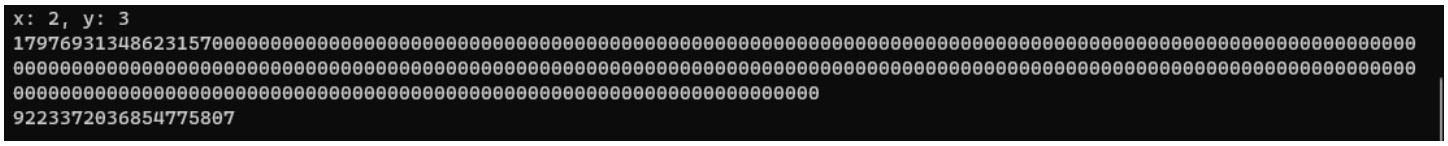
Floating-point numbers

File name: main.rs



```rust
fn main() {
    let x: f64 = 2.0; // f64
    let y: f32 = 3.0; // f32
    println!("x: {x}, y: {y}");
    //get max number
    println!("{}", std::f64::MAX); //
    println!("{}", std::i64::MAX); // 9223372036854775807

}
```

$ cargo run



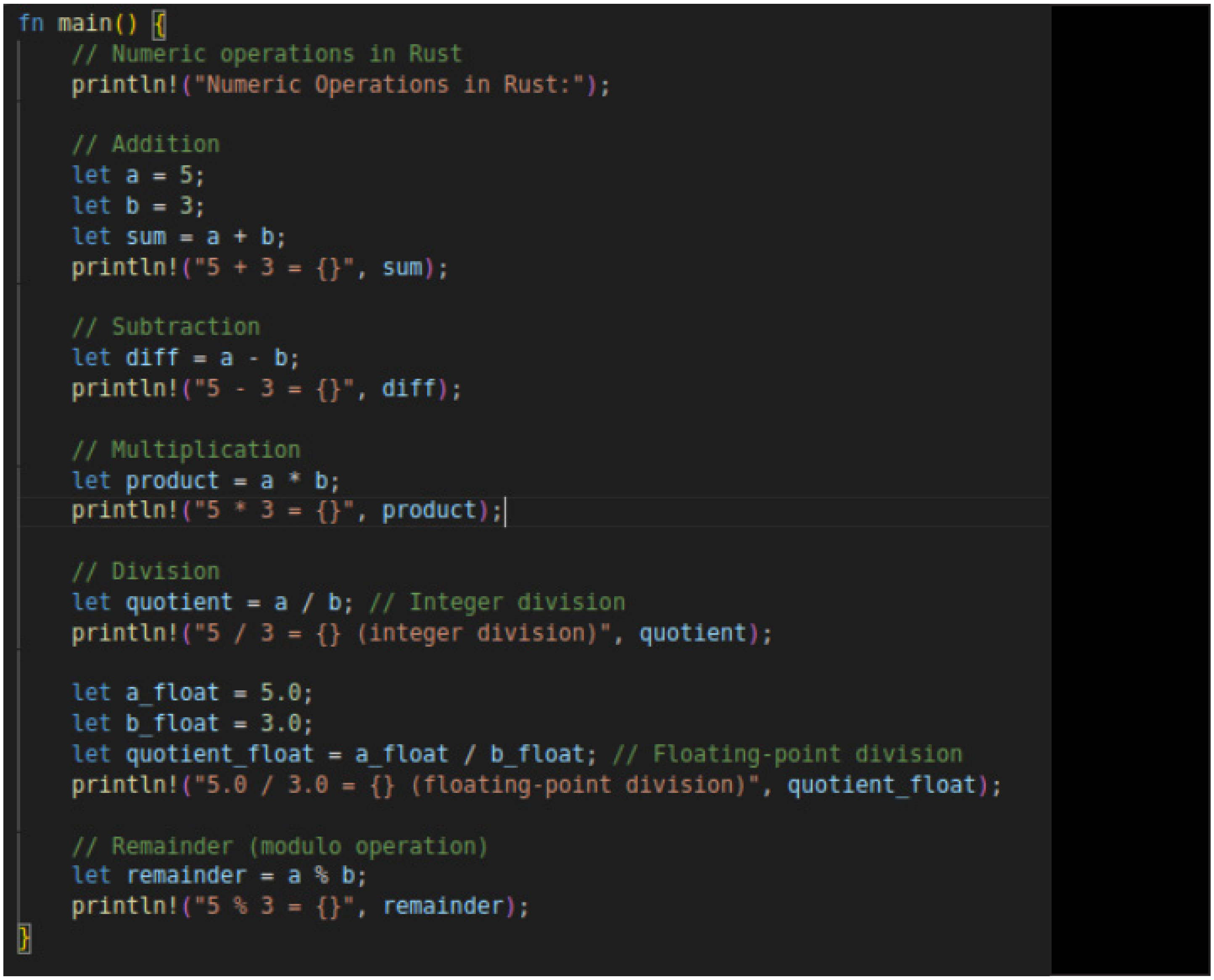
```
x: 2, y: 3
179769313486231570000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
9223372036854775807
```

Numeric Operations

Rust offers various numeric operations for number types, including addition, subtraction, multiplication, division, and remainder. Integer division truncates, rounding down to the nearest integer.

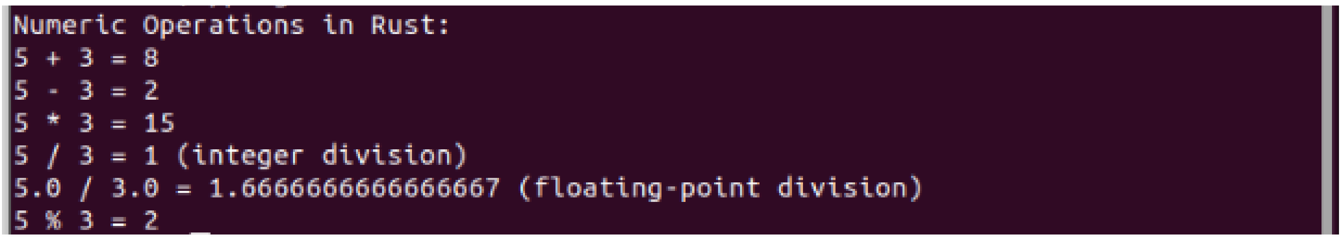
Numeric Operations

File name: main.rs



```rust
fn main() {
    // Numeric operations in Rust
    println!("Numeric Operations in Rust:");

    // Addition
    let a = 5;
    let b = 3;
    let sum = a + b;
    println!("5 + 3 = {}", sum);

    // Subtraction
    let diff = a - b;
    println!("5 - 3 = {}", diff);

    // Multiplication
    let product = a * b;
    println!("5 * 3 = {}", product);

    // Division
    let quotient = a / b; // Integer division
    println!("5 / 3 = {} (integer division)", quotient);

    let a_float = 5.0;
    let b_float = 3.0;
    let quotient_float = a_float / b_float; // Floating-point division
    println!("5.0 / 3.0 = {} (floating-point division)", quotient_float);

    // Remainder (modulo operation)
    let remainder = a % b;
    println!("5 % 3 = {}", remainder);
}
```

$ cargo run



```
Numeric Operations in Rust:
5 + 3 = 8
5 - 3 = 2
5 * 3 = 15
5 / 3 = 1 (integer division)
5.0 / 3.0 = 1.6666666666666667 (floating-point division)
5 % 3 = 2
```

Hexadecimals

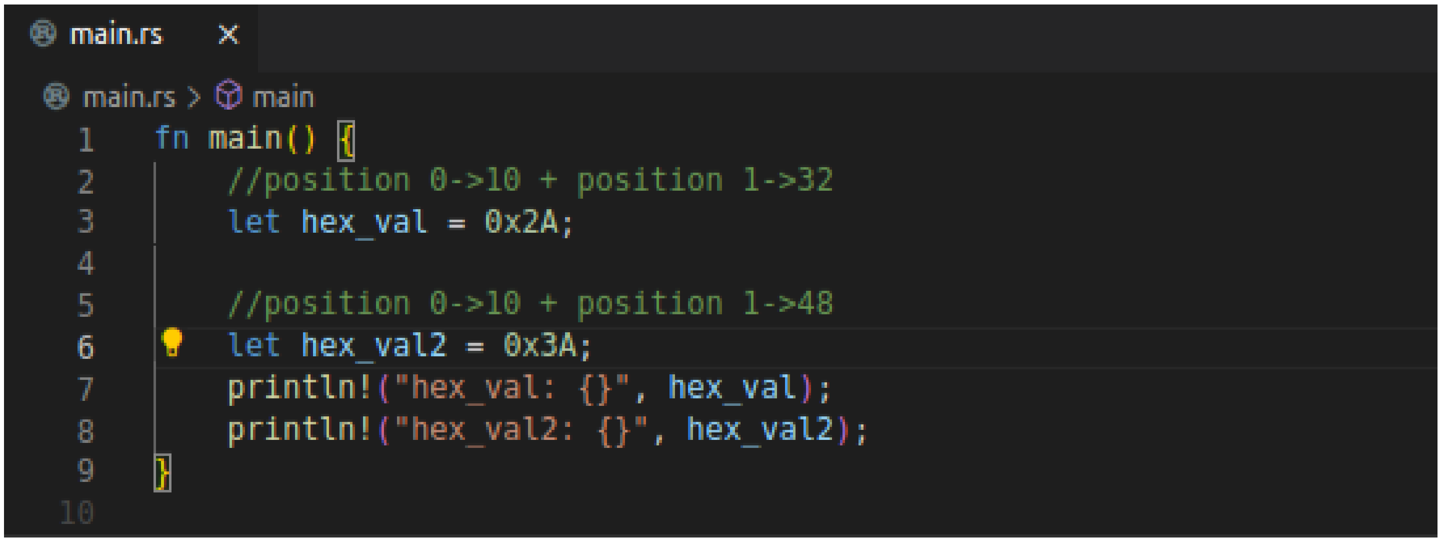
A hexadecimal is a base-16 number system representing numbers using 16 symbols: 0-9 for values 0-9, and A-F for values 10-15. It's important because it simplifies the representation and manipulation of binary data. Each hex digit can represent 4 bits, making it easier to read and write compared to binary.

For example, the binary string '1010' can be represented as the hex digit 'A.' In Rust, you write hexadecimals with the "0x" prefix.

This can be helpful when working with color codes, memory addresses, or bitwise operations.

Hexadecimals

File name: main.rs

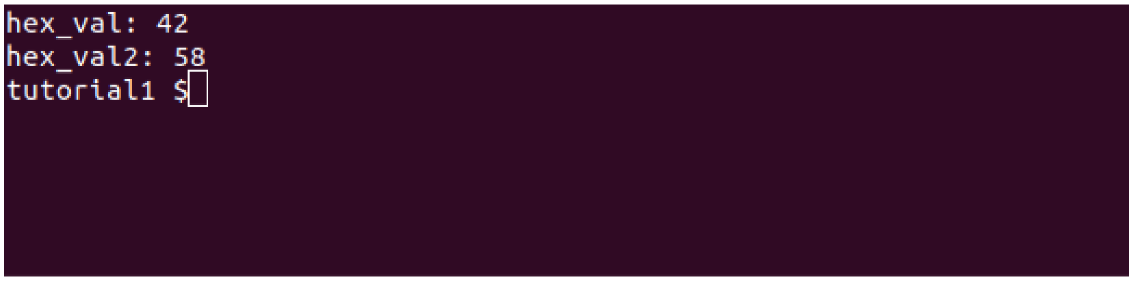


main.rs

main.rs > main

```rust
fn main() {
    //position 0->10 + position 1->32
    let hex_val = 0x2A;

    //position 0->10 + position 1->48
    let hex_val2 = 0x3A;
    println!("hex_val: {}", hex_val);
    println!("hex_val2: {}", hex_val2);
}
```

$ cargo run



```
hex_val: 42
hex_val2: 58
tutorial1 $
```

Octal numbers

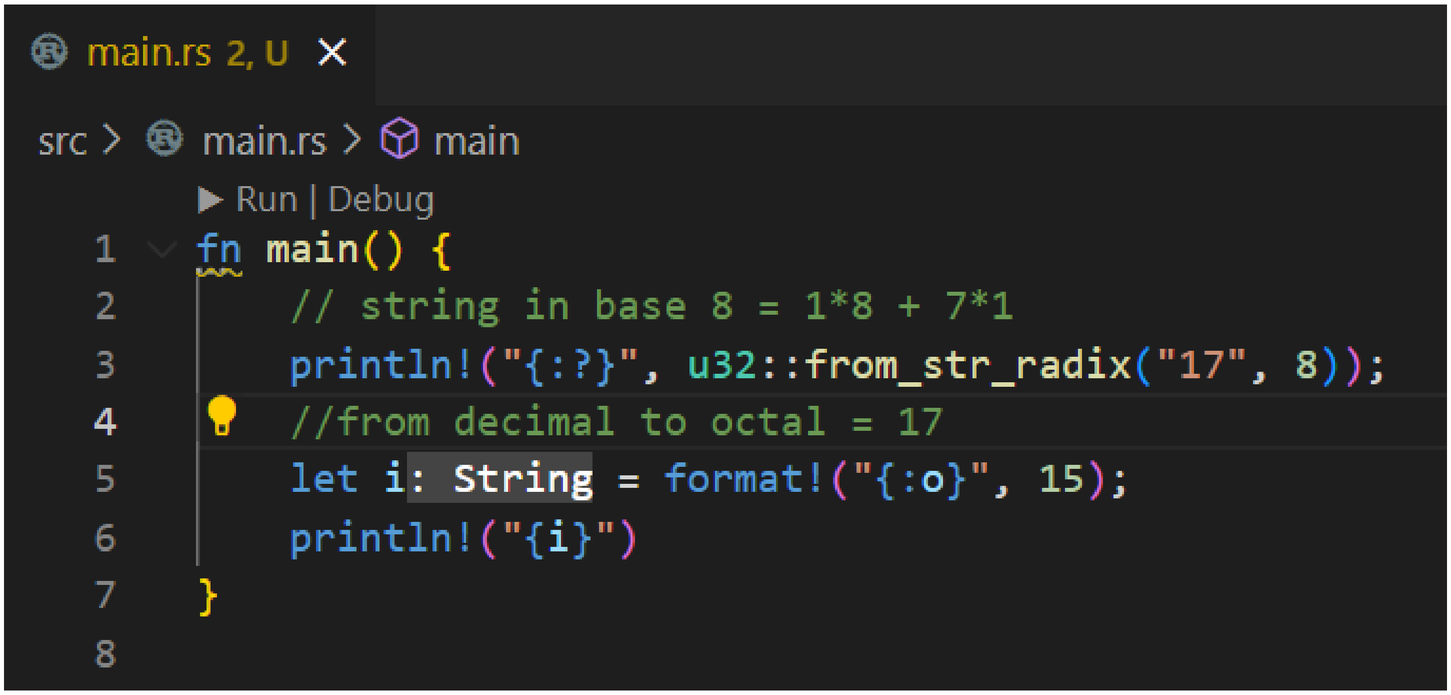
An octal number system comprises numbers represented using 8 unique digits: 0 to 7. It's essential for simplifying binary numbers and useful in computer programming for historical reasons. Octal numbers are written with an '0o' prefix, followed by base-8 digits.

Example: The octal value 0o572 corresponds to the decimal value (5 x 8^2) + (7 x 8^1) + (2 x 8^0) = 378.

In Rust, we can convert octal to decimal with `i32::from_str_radix`, and format decimal back to octal using `format!("{:o}", value)`.

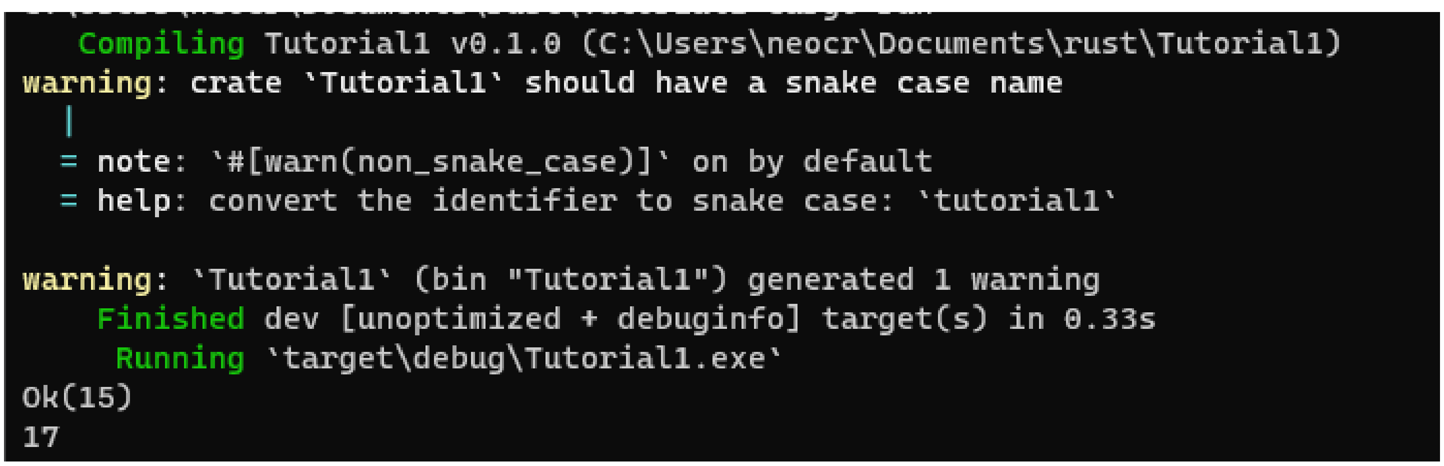
Octal numbers

File name: main.rs



```rust
fn main() {
    // string in base 8 = 1*8 + 7*1
    println!("{:?}", u32::from_str_radix("17", 8));
    //from decimal to octal = 17
    let i: String = format!("{:o}", 15);
    println!("{i}")
}

```

$ cargo run



```
   Compiling Tutorial1 v0.1.0 (C:\Users\neocr\Documents\rust\Tutorial1)
warning: crate `Tutorial1` should have a snake case name
  |
  = note: `#[warn(non_snake_case)]` on by default
  = help: convert the identifier to snake case: `tutorial1`

warning: `Tutorial1` (bin "Tutorial1") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.33s
     Running `target\debug\Tutorial1.exe`
Ok(15)
17
```

Variables and Mutability

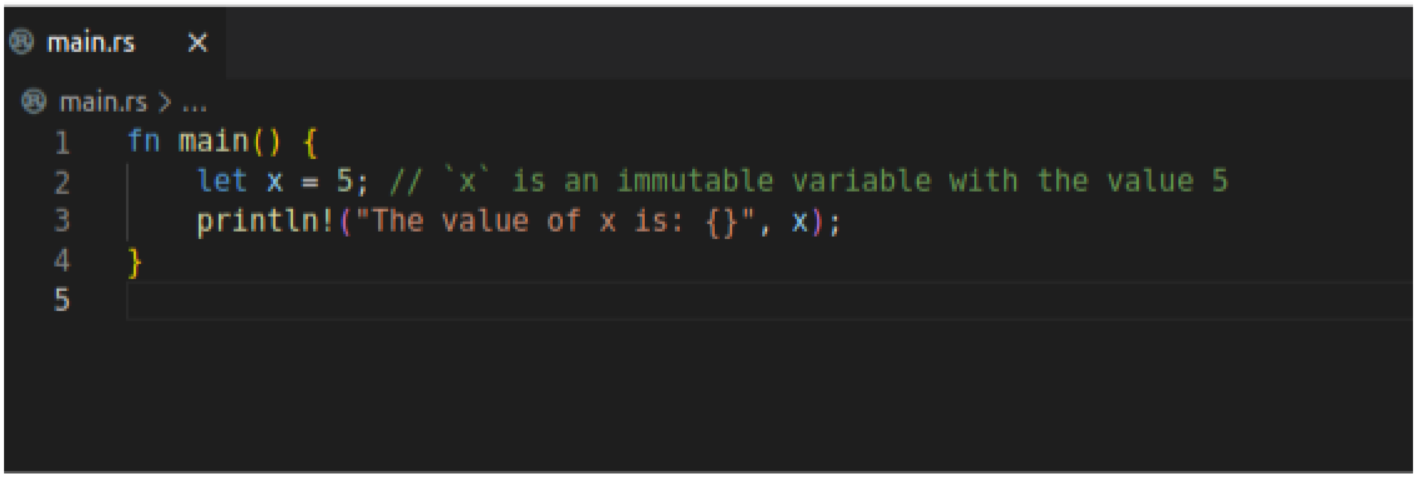
The `let` keyword
The `const` keyword
The `mut` keyword

The `let` keyword

let: The `let` keyword is used for creating and initializing a variable. By default, a variable declared with `let` is immutable, meaning its value cannot be changed after initialization. This enforces safety and security in your code.

The `let` keyword

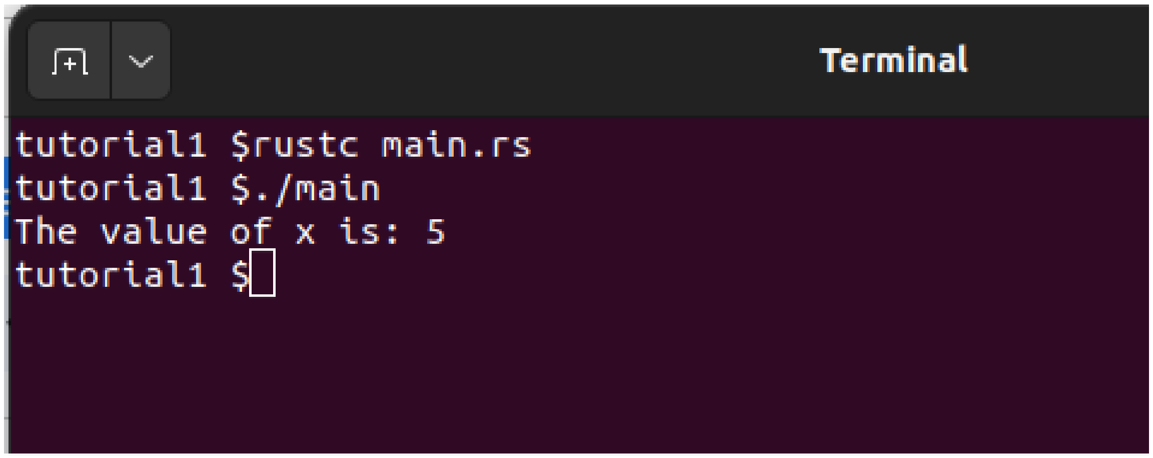
File name: main.rs



```rust
fn main() {
    let x = 5; // `x` is an immutable variable with the value 5
    println!("The value of x is: {}", x);
}
```

Value can be reassigned (this is not possible with `const` keyword):

$ cargo run



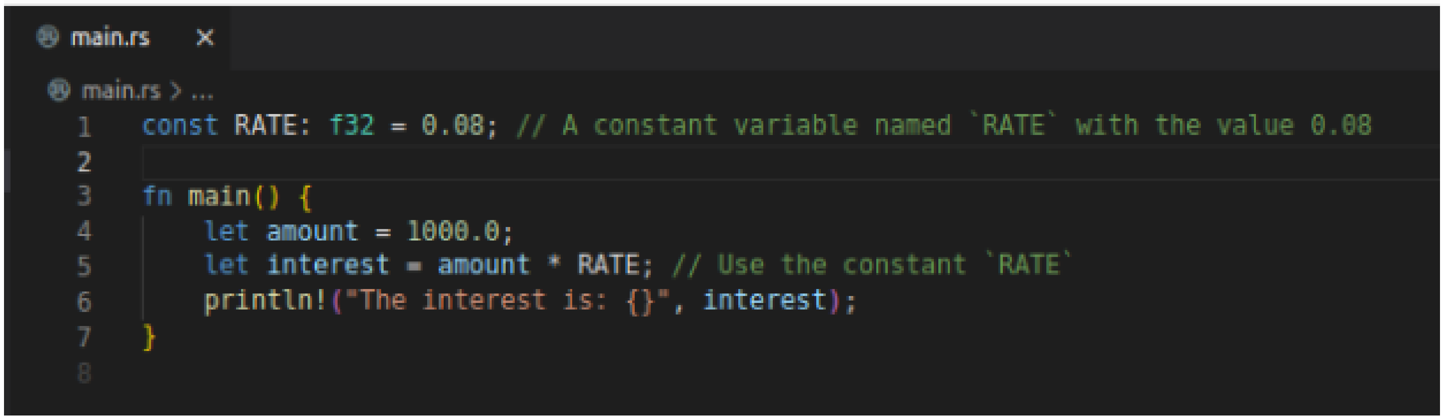
```
tutorial1 $rustc main.rs
tutorial1 $./main
The value of x is: 5
tutorial1 $
```

The `const` keyword

const: The `const` keyword is similar to `let`, but is used to declare a constant variable. This means its value never changes and must be initialized with a constant expression (not a value computed at runtime). Constants are always immutable and have a global scope.

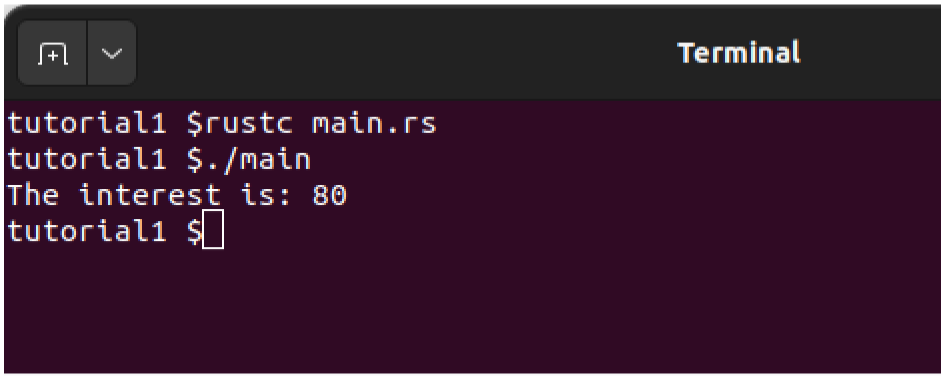
The `const` keyword

File name: main.rs



```
main.rs  ×
main.rs > ...
const RATE: f32 = 0.08; // A constant variable named `RATE` with the value 0.08

fn main() {
    let amount = 1000.0;
    let interest = amount * RATE; // Use the constant `RATE`
    println!("The interest is: {}", interest);
}
```

$ cargo run



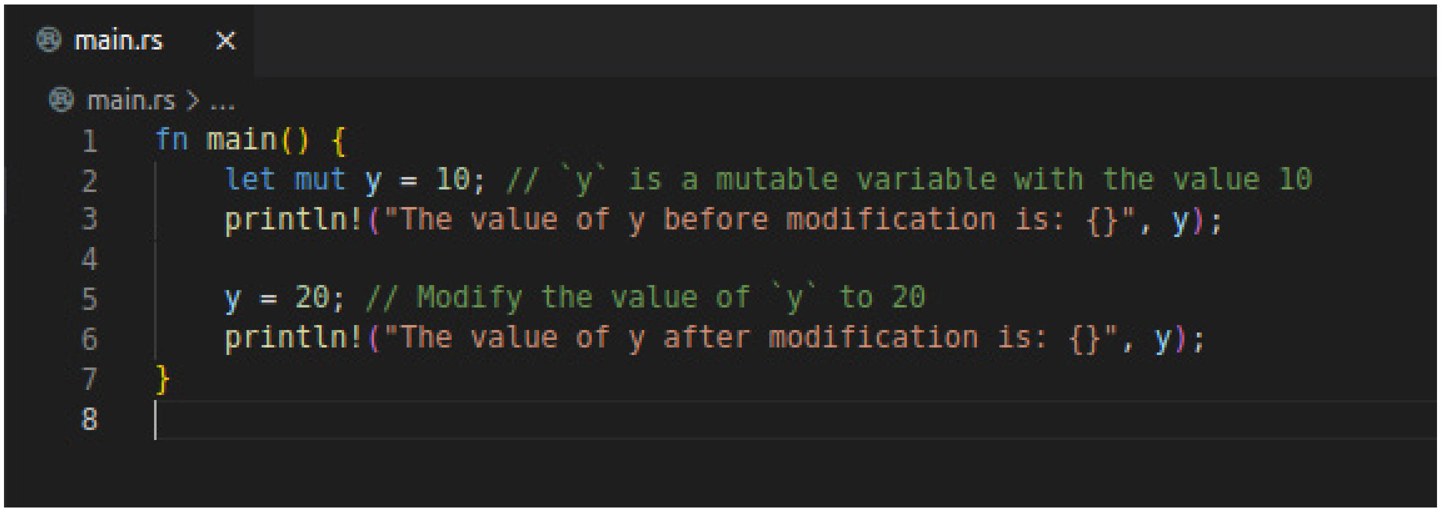
```
tutorial1 $rustc main.rs
tutorial1 $./main
The interest is: 80
tutorial1 $
```

The `**mut**` keyword

mut: The `mut` keyword is used to make a variable mutable, allowing its value to change after initialization. This is useful when working with data that needs to be modified during program execution.

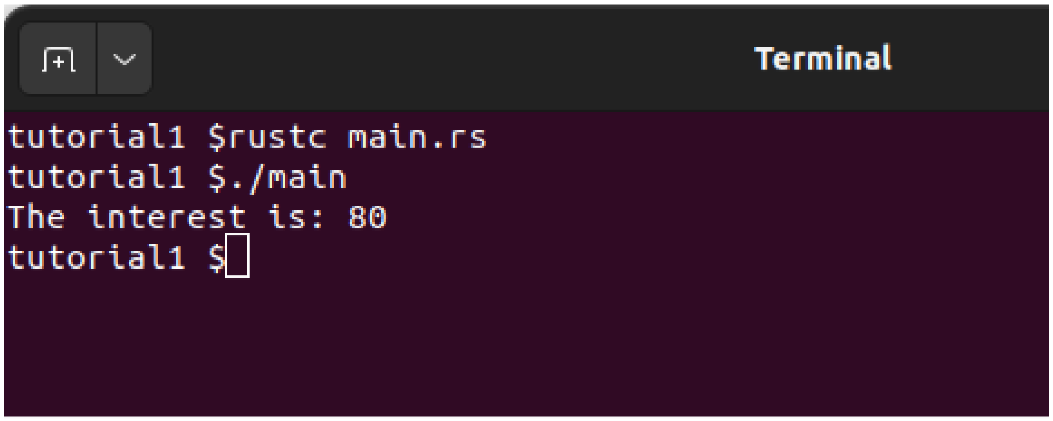
The `mut` keyword

File name: main.rs



```rust
fn main() {
    let mut y = 10; // `y` is a mutable variable with the value 10
    println!("The value of y before modification is: {}", y);

    y = 20; // Modify the value of `y` to 20
    println!("The value of y after modification is: {}", y);
}

```

$ cargo run



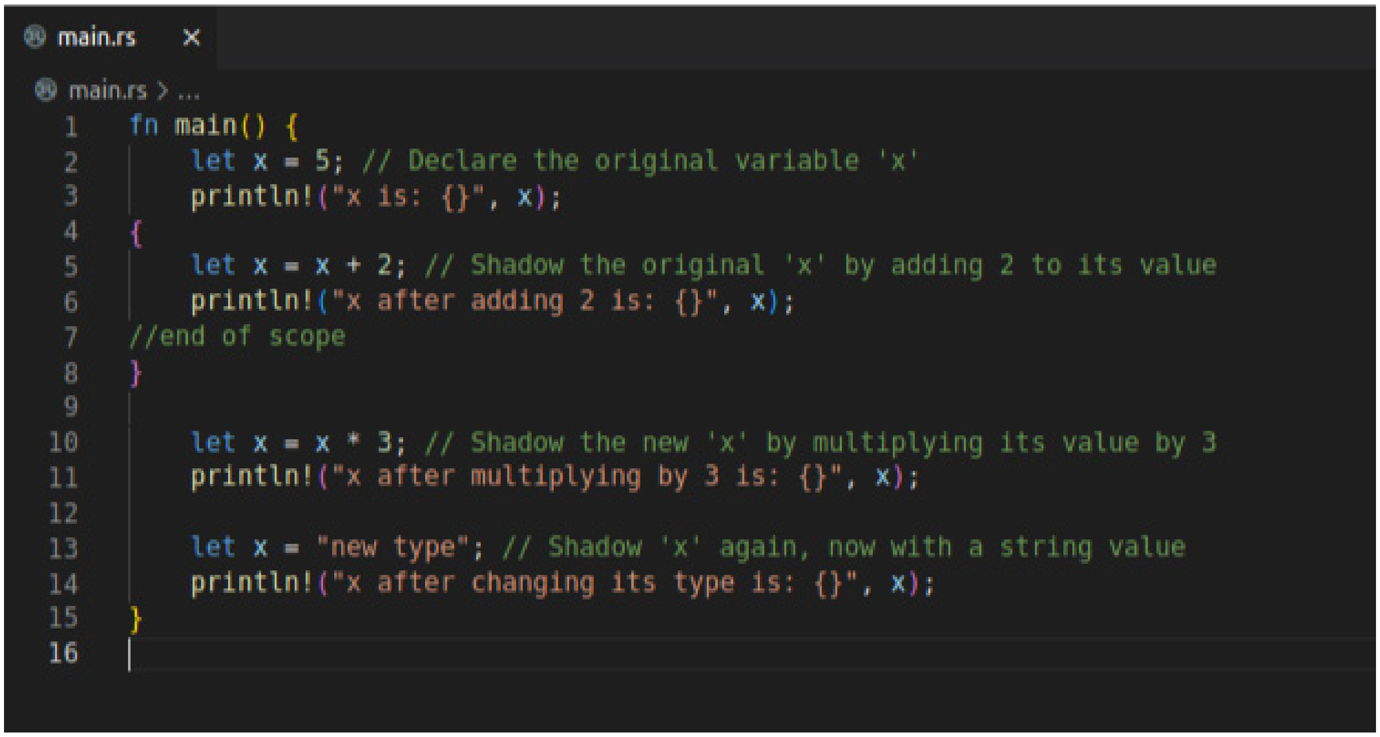
```
tutorial1 $rustc main.rs
tutorial1 $./main
The interest is: 80
tutorial1 $
```

Shadowing

Shadowing is a concept in Rust programming where you declare a new variable with the same name as an existing variable, effectively "shadowing" the original variable. This allows you to change the value and type of a variable by shadowing it, without affecting the original value. This can be useful in cases where you need to perform transformations on a variable while keeping the same name.

Shadowing

File name: main.rs

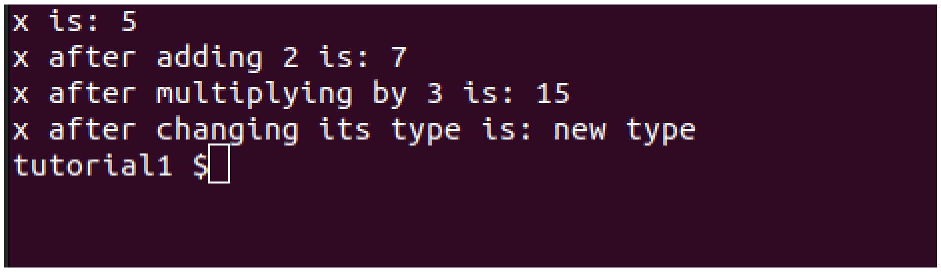


main.rs ×

main.rs > ...

```rust
fn main() {
    let x = 5; // Declare the original variable 'x'
    println!("x is: {}", x);
{
    let x = x + 2; // Shadow the original 'x' by adding 2 to its value
    println!("x after adding 2 is: {}", x);
//end of scope
}

    let x = x * 3; // Shadow the new 'x' by multiplying its value by 3
    println!("x after multiplying by 3 is: {}", x);

    let x = "new type"; // Shadow 'x' again, now with a string value
    println!("x after changing its type is: {}", x);
}
```

$ cargo run



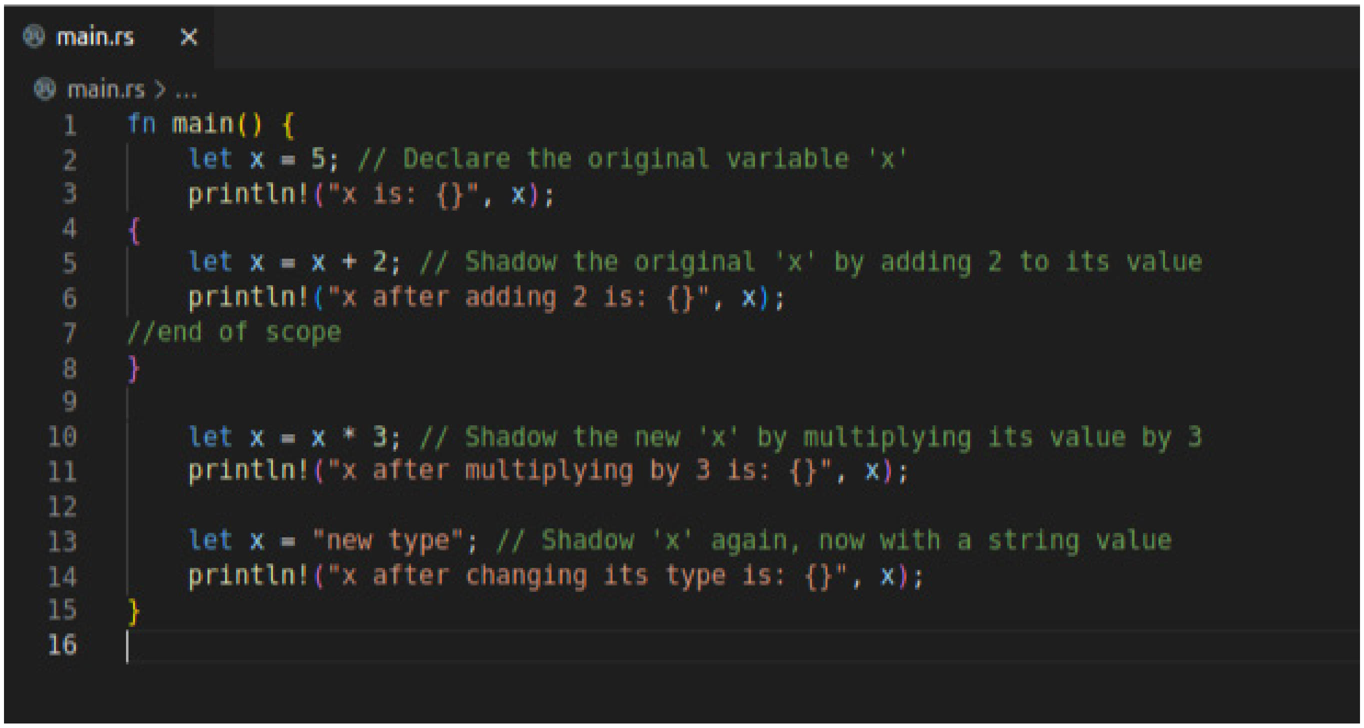
```
x is: 5
x after adding 2 is: 7
x after multiplying by 3 is: 15
x after changing its type is: new type
tutorial1 $
```

Booleans (The Boolean Type)
Booleans, in the context of programming, are a data type that can represent one of two possible values: true or false. They are named after George Boole, a mathematician who developed Boolean algebra, an essential concept in computer science.

In Rust programming, Booleans are represented by the `bool` data type. They are commonly used in conditional statements, such as `if`, `else`, and `while`, to control the flow of the program's execution based on specific conditions.

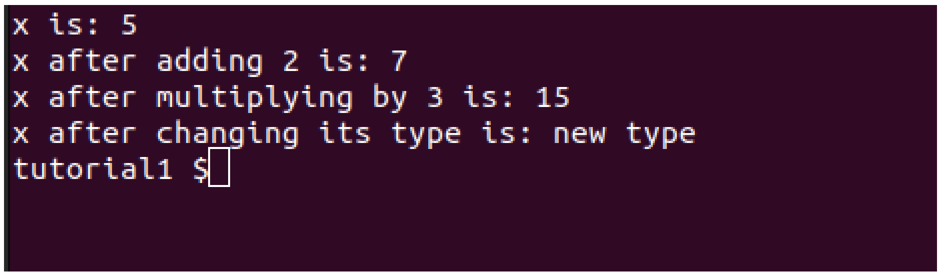
Booleans

File name: main.rs



```rust
fn main() {
    let x = 5; // Declare the original variable 'x'
    println!("x is: {}", x);
{
    let x = x + 2; // Shadow the original 'x' by adding 2 to its value
    println!("x after adding 2 is: {}", x);
//end of scope
}

    let x = x * 3; // Shadow the new 'x' by multiplying its value by 3
    println!("x after multiplying by 3 is: {}", x);

    let x = "new type"; // Shadow 'x' again, now with a string value
    println!("x after changing its type is: {}", x);
}
```

$ cargo run



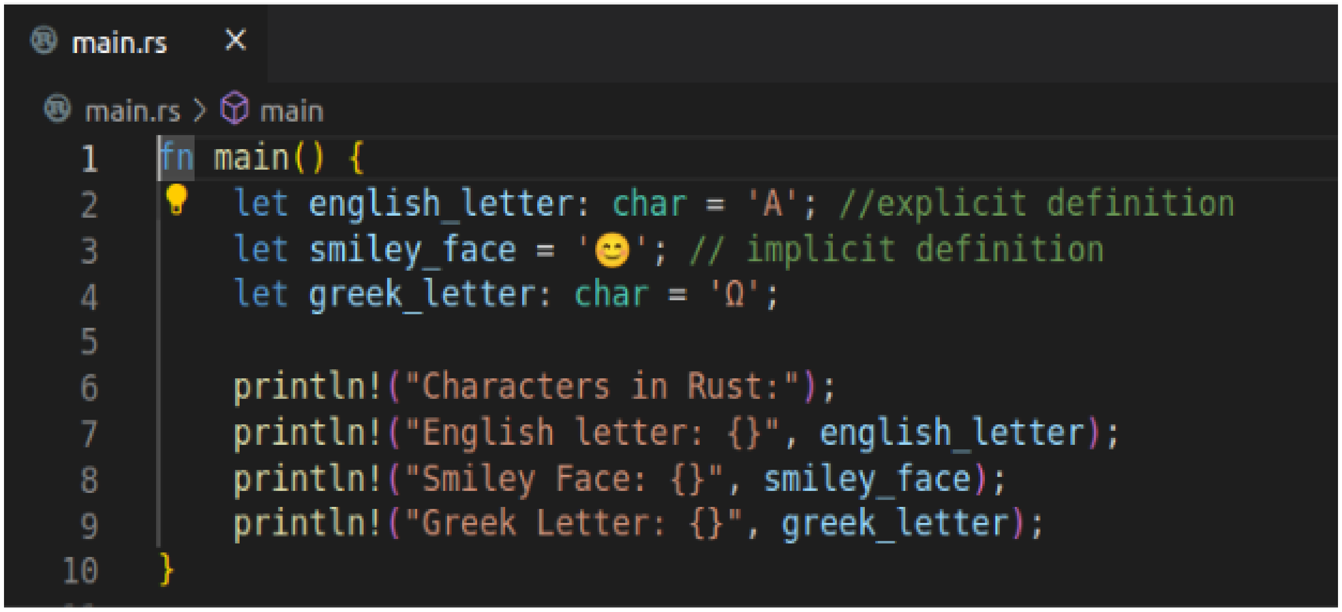
```
x is: 5
x after adding 2 is: 7
x after multiplying by 3 is: 15
x after changing its type is: new type
tutorial1 $
```

Characters in Rust The Character Type
A “Character Type” in Rust refers to the data type that represents an individual Unicode character. It is denoted by the keyword `char`, and it can store a single Unicode character, taking up 4 bytes of memory. Unicode characters include not only standard ASCII characters (like digits, English letters, and common symbols) but also a wide
range of international characters, special symbols, and emojis.
Accented letters in several languages; Japanese, Chinese, and Korean; emojis, and zero-width spaces are all valid char values in Rust

Here’s a code example to illustrate how to work with the character type in Rust:

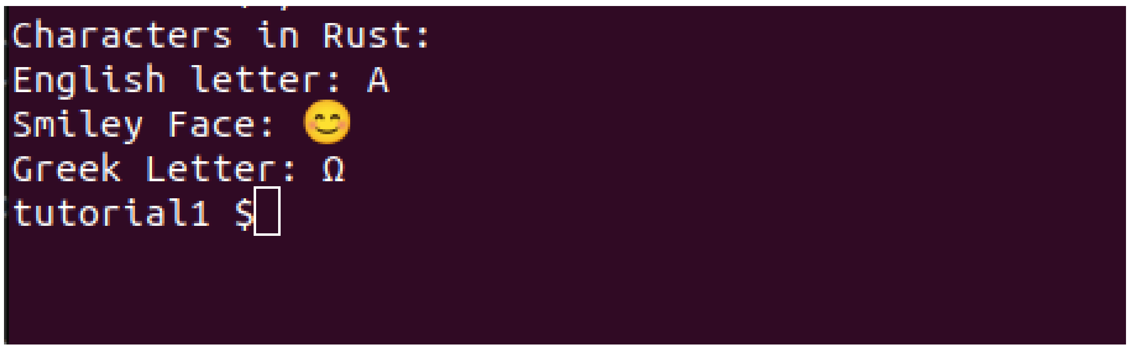
Characters

File name: main.rs



```rust
fn main() {
    let english_letter: char = 'A'; //explicit definition
    let smiley_face = '😊'; // implicit definition
    let greek_letter: char = 'Ω';

    println!("Characters in Rust:");
    println!("English letter: {}", english_letter);
    println!("Smiley Face: {}", smiley_face);
    println!("Greek Letter: {}", greek_letter);
}
```

$ cargo run



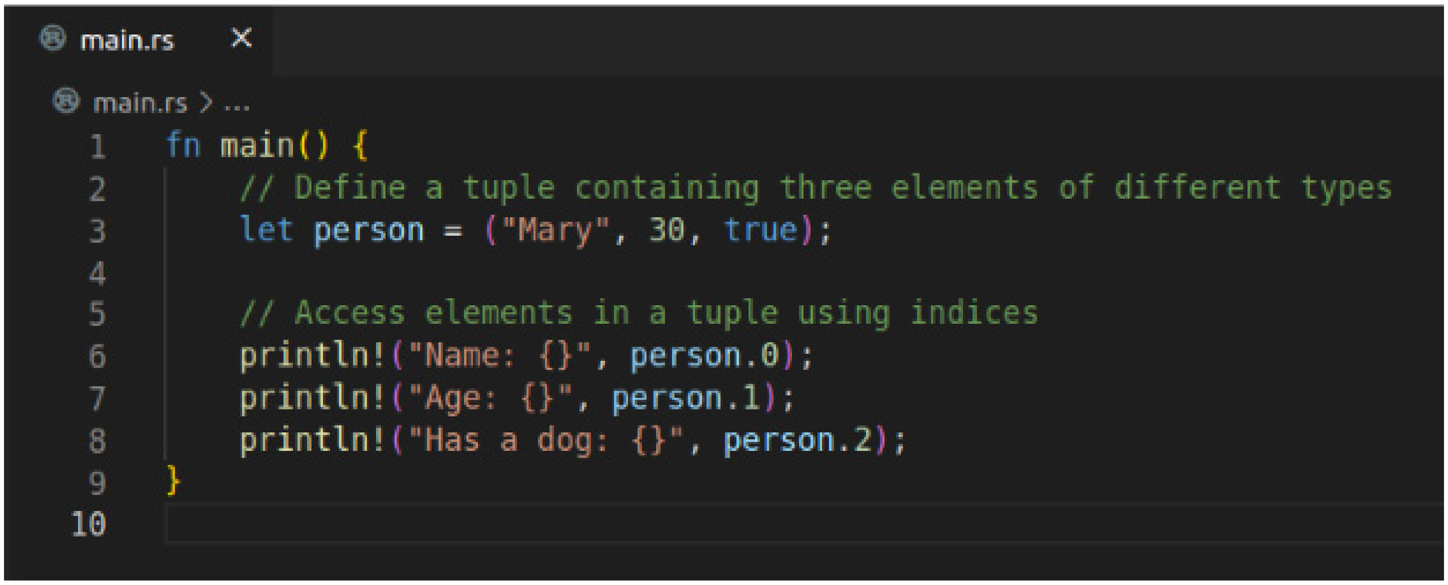
```
Characters in Rust:
English letter: A
Smiley Face: 😊
Greek Letter: Ω
tutorial1 $
```

Tuple Type

Tuple type is a compound data structure in Rust that allows you to bundle multiple heterogenous (values with different types) together as a single entity, with a fixed length. You can think of tuples as a simple way to create a collection of differing types, while keeping the data organized and easily accessible through in-dexing.

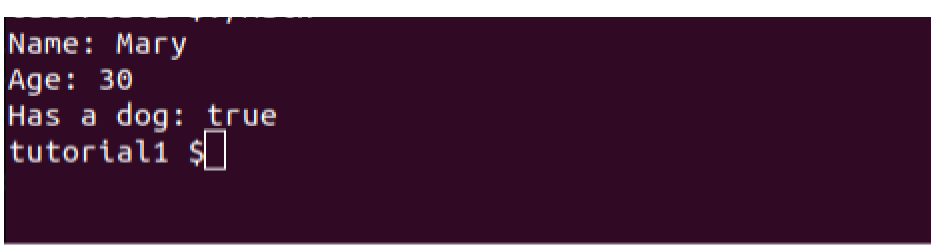
Tuple Type

File name: main.rs



```rust
fn main() {
    // Define a tuple containing three elements of different types
    let person = ("Mary", 30, true);

    // Access elements in a tuple using indices
    println!("Name: {}", person.0);
    println!("Age: {}", person.1);
    println!("Has a dog: {}", person.2);
}
```

$ cargo run

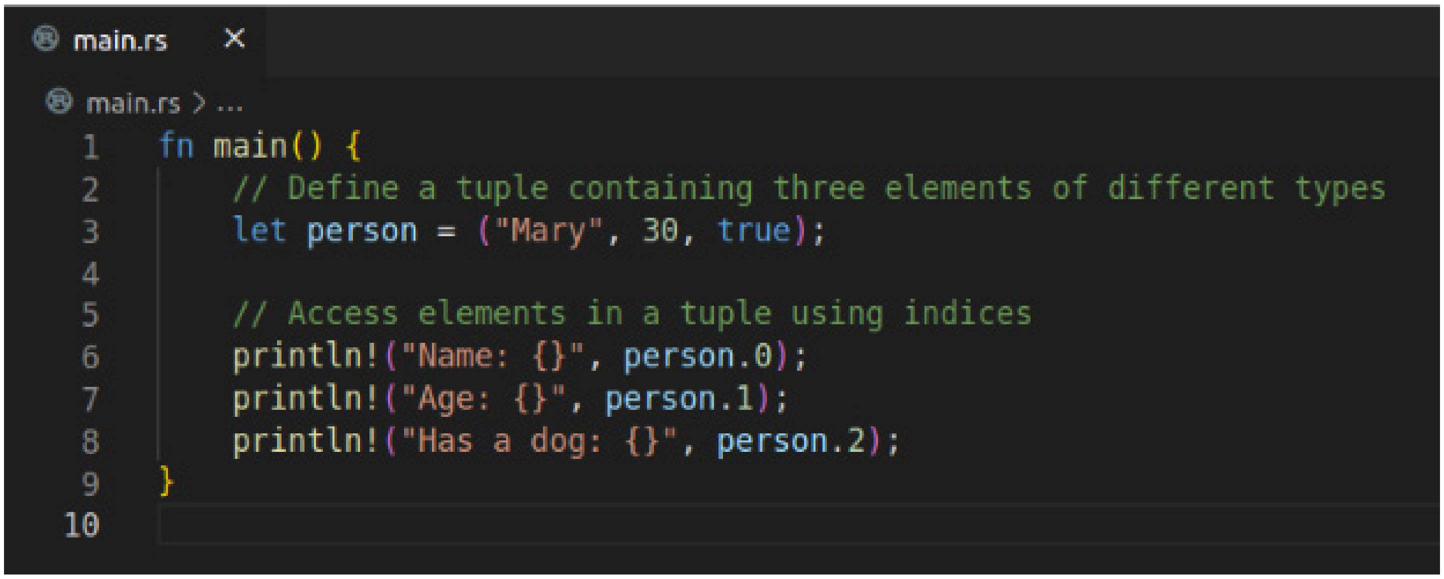


```
Name: Mary
Age: 30
Has a dog: true
tutorial1 $
```

Tuple Type

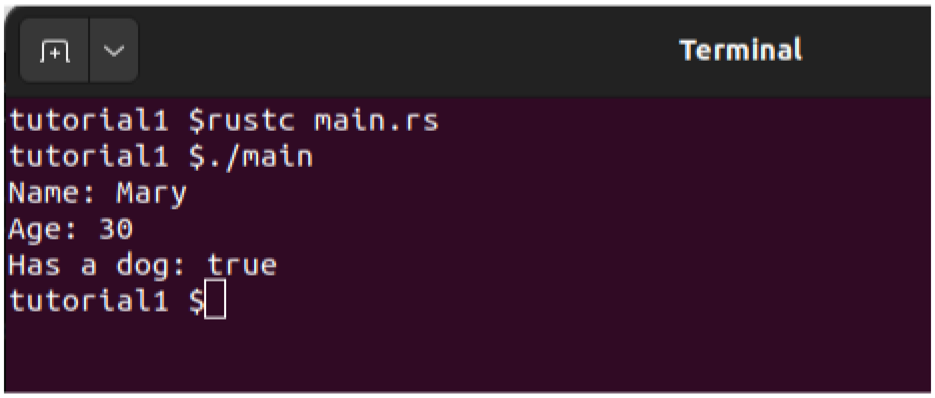
Tuple Type

File name: main.rs



```rust
fn main() {
    // Define a tuple containing three elements of different types
    let person = ("Mary", 30, true);

    // Access elements in a tuple using indices
    println!("Name: {}", person.0);
    println!("Age: {}", person.1);
    println!("Has a dog: {}", person.2);
}
```

$ cargo run



```
tutorial1 $rustc main.rs
tutorial1 $./main
Name: Mary
Age: 30
Has a dog: true
tutorial1 $
```

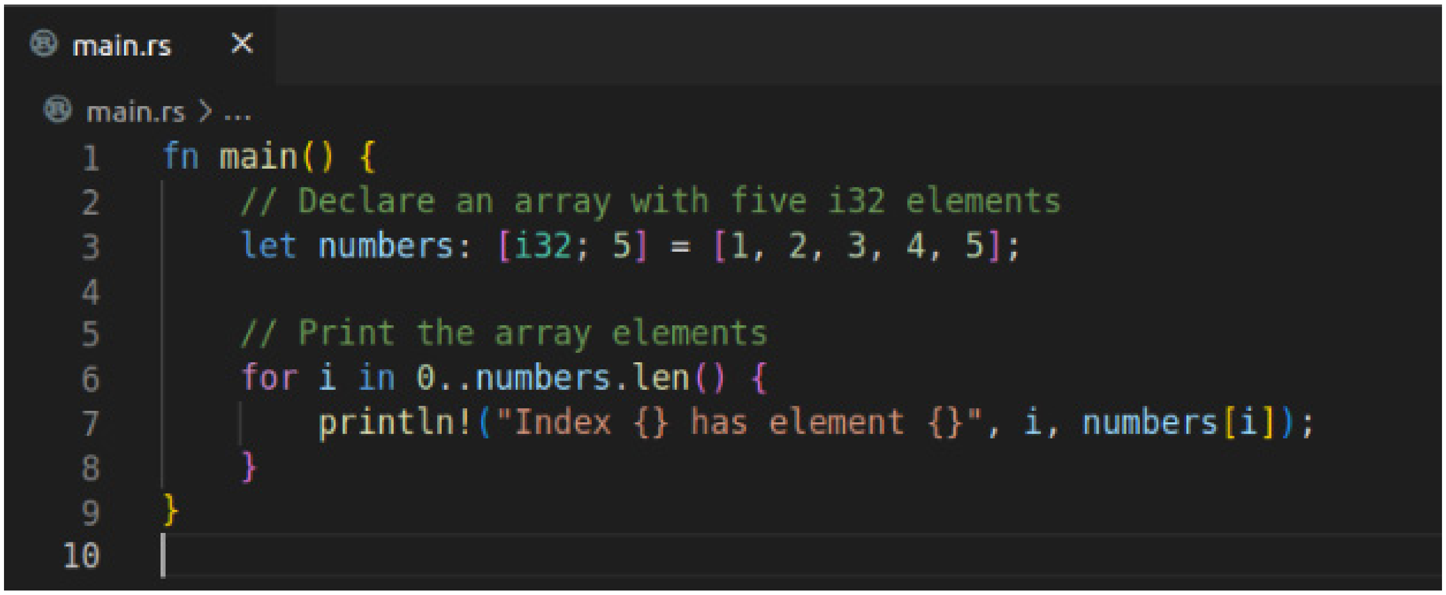
Arrays

An “Array Type” is a fixed-size, homogeneous data structure in Rust programming language that can store multiple values of the same data type. In simpler terms, it’s a collection of elements where each element is of the same type, and the number of elements is fixed, meaning the size of the array is constant.

Here’s a code example to demonstrate the use of an array type in Rust:

Arrays

File name: main.rs

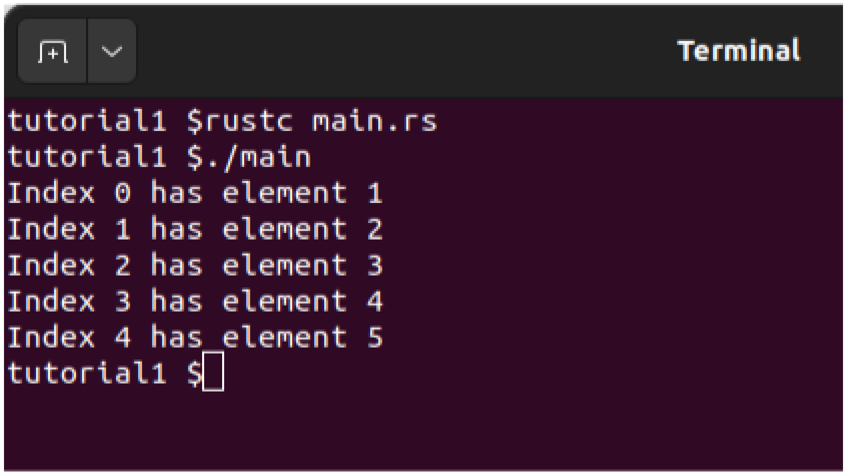


main.rs ✕

main.rs > ...

```rust
fn main() {
    // Declare an array with five i32 elements
    let numbers: [i32; 5] = [1, 2, 3, 4, 5];

    // Print the array elements
    for i in 0..numbers.len() {
        println!("Index {} has element {}", i, numbers[i]);
    }
}
```

$ cargo run



Terminal

```
tutorial1 $rustc main.rs
tutorial1 $./main
Index 0 has element 1
Index 1 has element 2
Index 2 has element 3
Index 3 has element 4
Index 4 has element 5
tutorial1 $
```

Functions

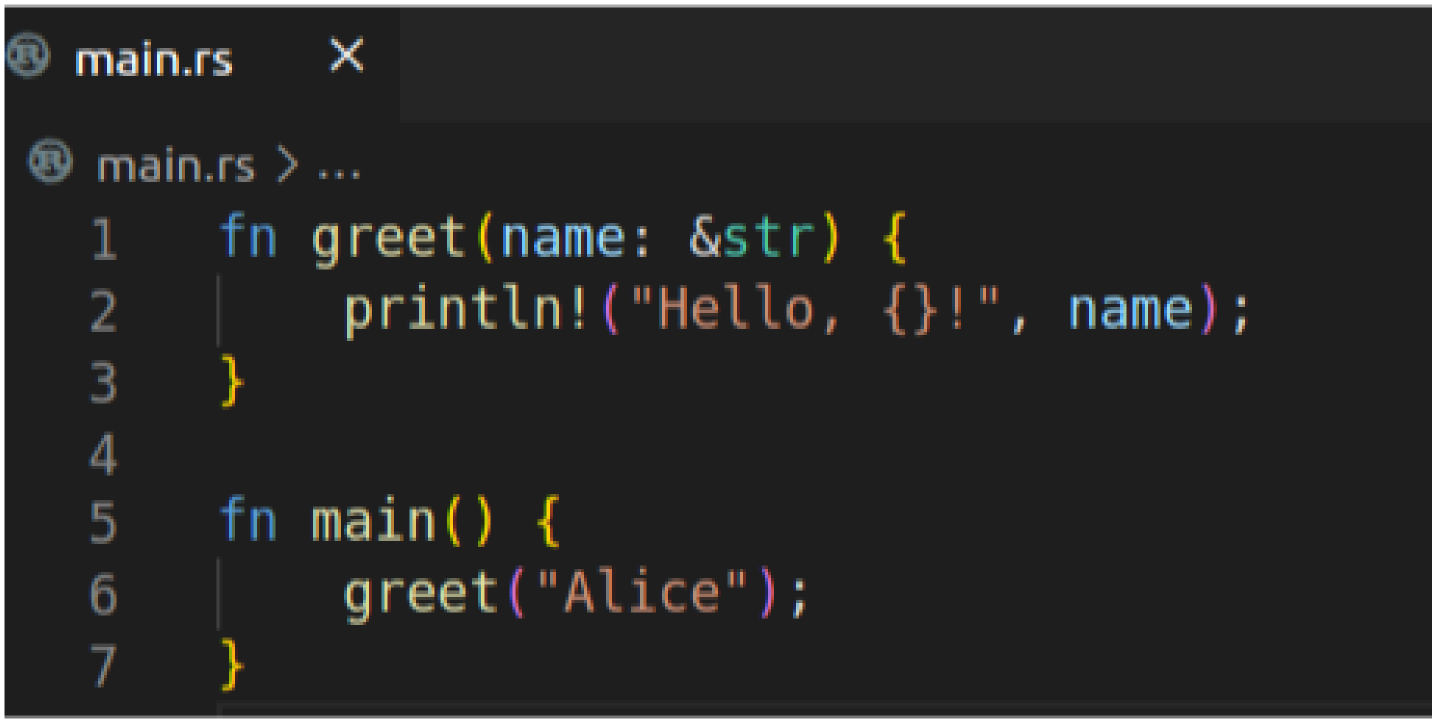
Functions in Rust are reusable pieces of code that perform a specific task. They are defined with the “**fn**” keyword, followed by the function’s name, a list of input parameters, and the return type of the function if applicable. Functions allow us to modularize, make our code more organized, and prevent repetition by abstracting common functionality into one place.

Arguments and parameters are essential parts of a function. Parameters are the variables listed in the function definition, while arguments are the values passed to a function when it’s called. Simply put, parameters are the placeholders for the data a function needs, and arguments hold the actual data being supplied to the function.

In Rust, it is a convention to name functions using the lower snake case. This means separating words with underscores and using lowercase letters for the words (example: calculate_sum).

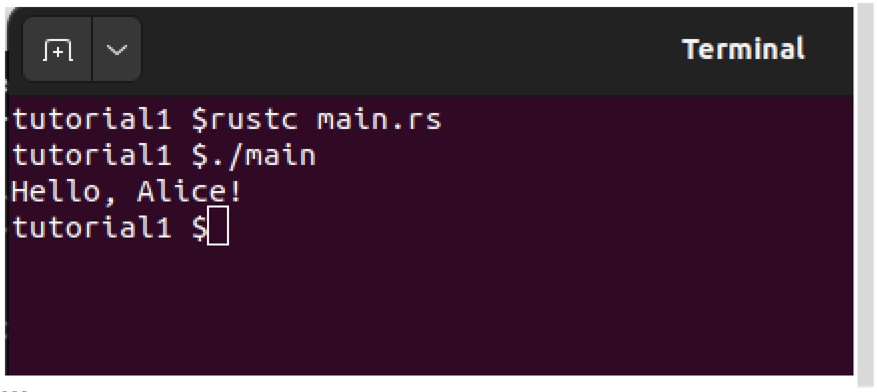
Functions

File name: main.rs



```rust
fn greet(name: &str) {
    println!("Hello, {}!", name);
}

fn main() {
    greet("Alice");
}
```

$ cargo run



```
tutorial1 $rustc main.rs
tutorial1 $./main
Hello, Alice!
tutorial1 $
```

Cargo

Cargo is the default package manager and build tool for the Rust programming language. It helps developers manage their project’s dependencies, compile source code, run tests, and distribute their compiled applications easily. Essentially, Cargo streamlines the entire development process in Rust, allowing programmers to focus more on coding.

Building Cargo consists of several commands, including:

1. `**cargo build**`: This command compiles your Rust project and generates an executable binary in the "target/debug" directory. It also downloads and compiles all dependencies specified in the "Cargo.toml" file.

2. `**cargo run**`: This command not only builds the project but also immediately runs the compiled binary. It is particularly useful during development because it simplifies the process of testing your code.

`**cargo check**` is another useful command that analyzes your source code and determines if it has any syntax or type errors without actually building it. This works much faster than the `cargo build` command because it does not create an executable binary, which can save developers a significant amount of time during the development process.

Building for release is an essential step in the deployment of a Rust application. To optimize your code for better performance, use the command `**cargo build --release**`. This prepares your project with optimizations, creating a more efficient binary in the "***target/release***" directory that is ideal for distribution.

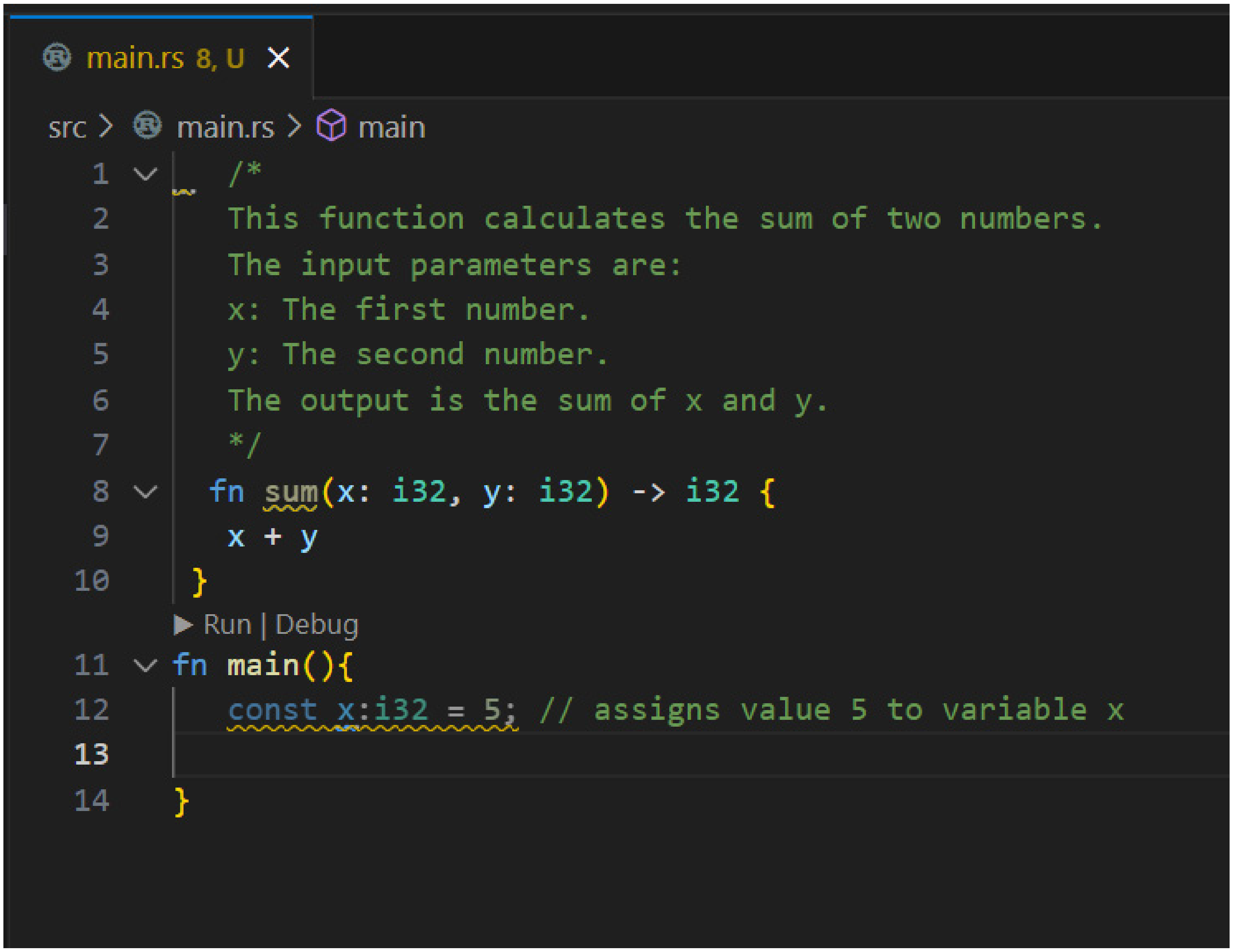
1. Line Comments: Start line comments with `//` followed by the explanation. Use line comments to describe a single line of code or a specific part of an expression. Ensure they are concise and direct.

   Example:
   ```rust
   let x = 5; // assigns value 5 to variable x
   ```

2. Block Comments: Enclose block comments between `/*` and `*/`. Block comments should be used to add more detailed descriptions that span multiple lines or to comment out a block of temporary code.

Comments

File name: main.rs



```rust
/*
This function calculates the sum of two numbers.
The input parameters are:
x: The first number.
y: The second number.
The output is the sum of x and y.
*/
 fn sum(x: i32, y: i32) -> i32 {
  x + y
}
fn main(){
  const x:i32 = 5; // assigns value 5 to variable x

}
```

Control Flow

Control flow refers to the order in which the code statements are executed, allowing programmers to build conditional branches, loops, and other structures to perform tasks. In Rust, control flow elements primarily include if/else if expressions, while loops, and for loops.

Control Flow

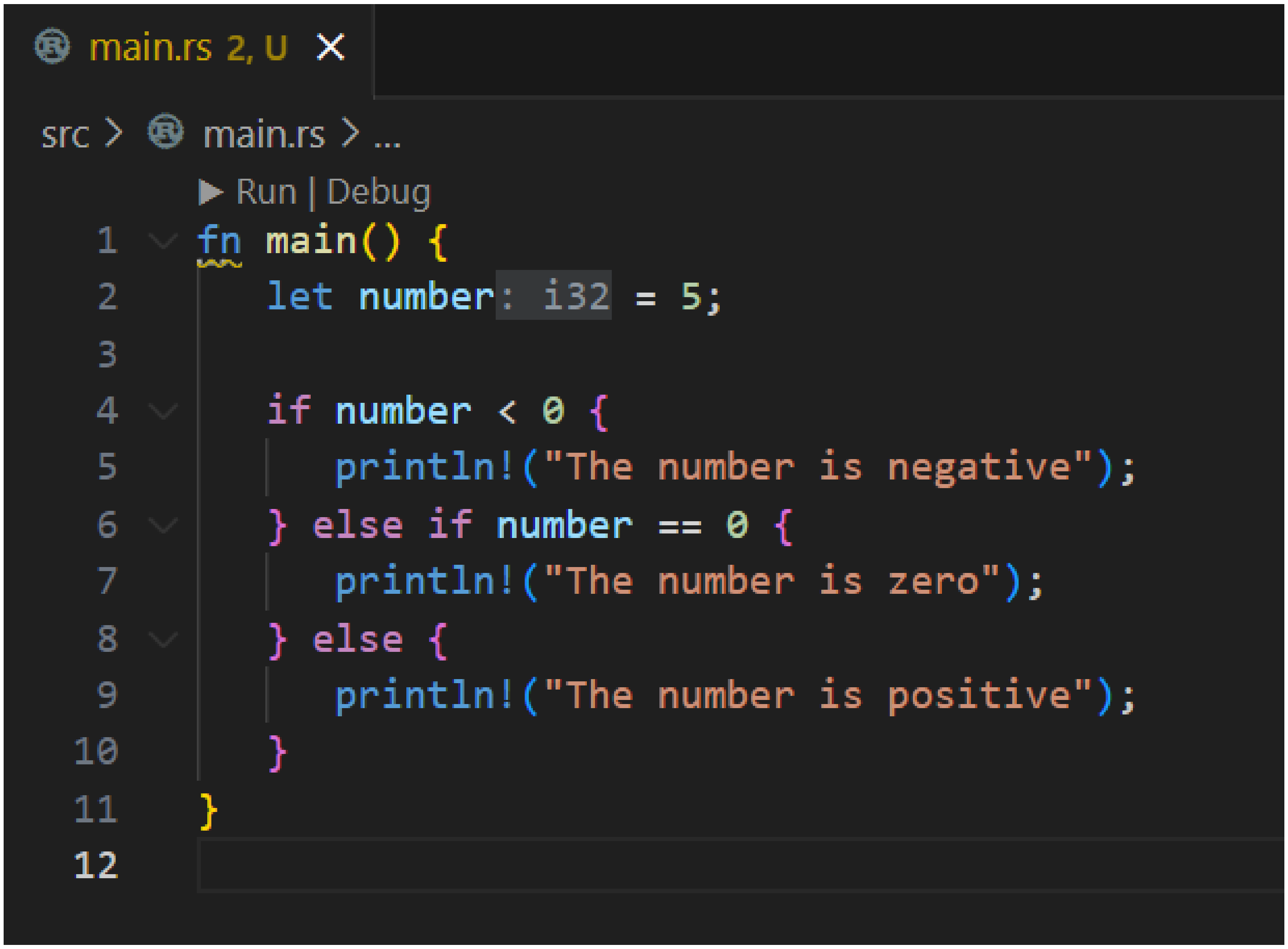
- if/else if expressions
- while loops
- and for loops.
- for loops

If-else if expressions

1. If-else if expressions: These are used to execute code depending on specific conditions. The if keyword is followed by a condition, while the else-if keyword can be used to define additional conditions if the previous conditions are not met. If no conditions are met, the else keyword can be used to define a fallback action.

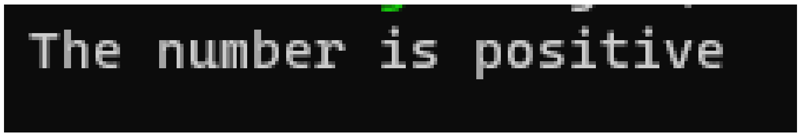
If-else if expressions

File name: main.rs



```rust
fn main() {
    let number: i32 = 5;

    if number < 0 {
        println!("The number is negative");
    } else if number == 0 {
        println!("The number is zero");
    } else {
        println!("The number is positive");
    }
}
```

$ cargo run

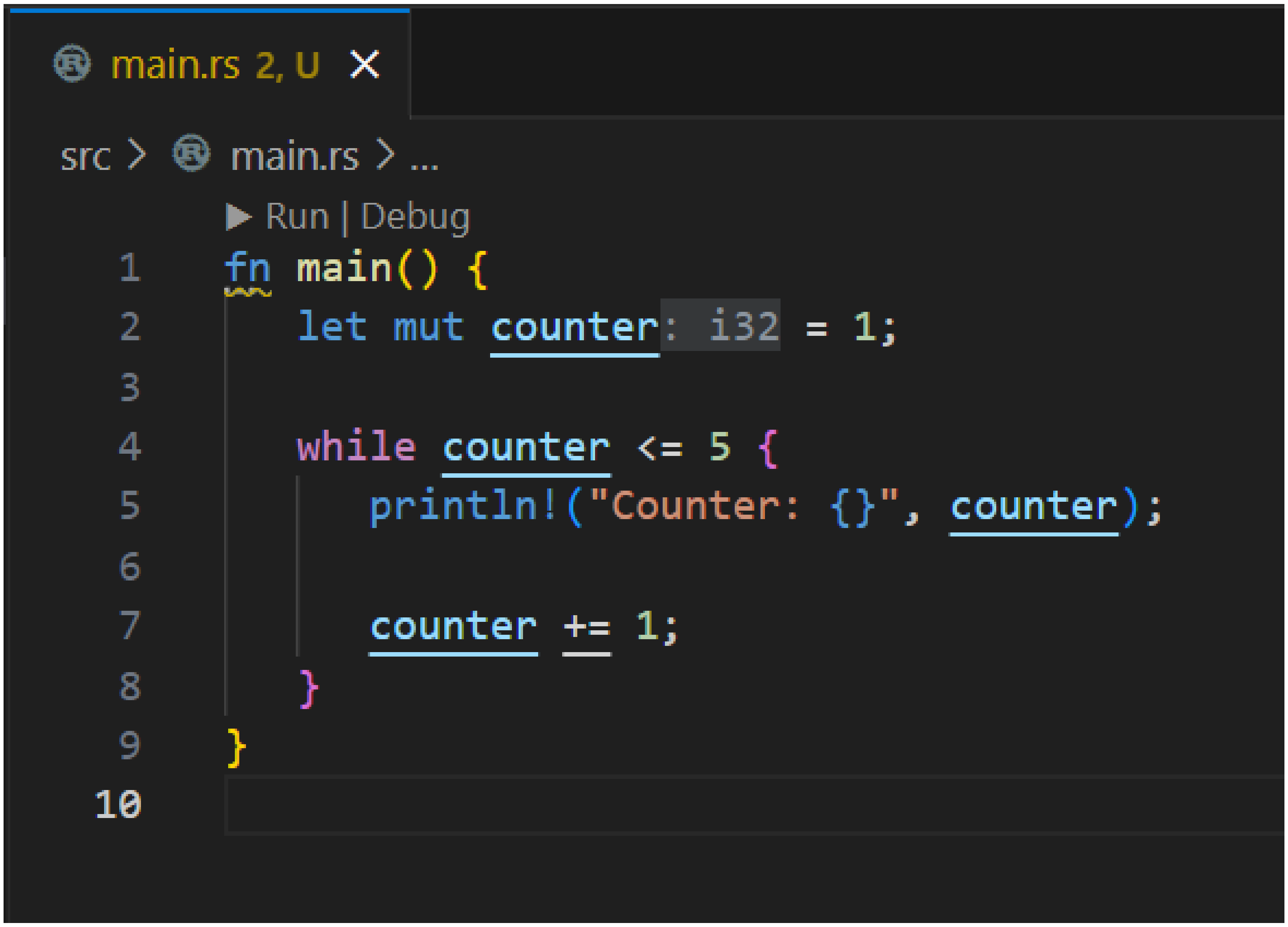


```
The number is positive
```

2. While loop: A while loop allows you to execute a block of code repeatedly as long as the specified condition remains true. The loop will continue until the condition is false or a break statement is encountered.

While loop

File name: main.rs

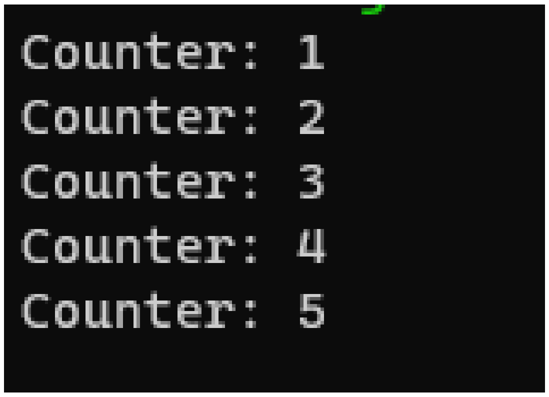

main.rs 2, U ×

src > main.rs > ...

▶ Run | Debug

```rust
fn main() {
    let mut counter: i32 = 1;

    while counter <= 5 {
        println!("Counter: {}", counter);

        counter += 1;
    }
}
```

$ cargo run

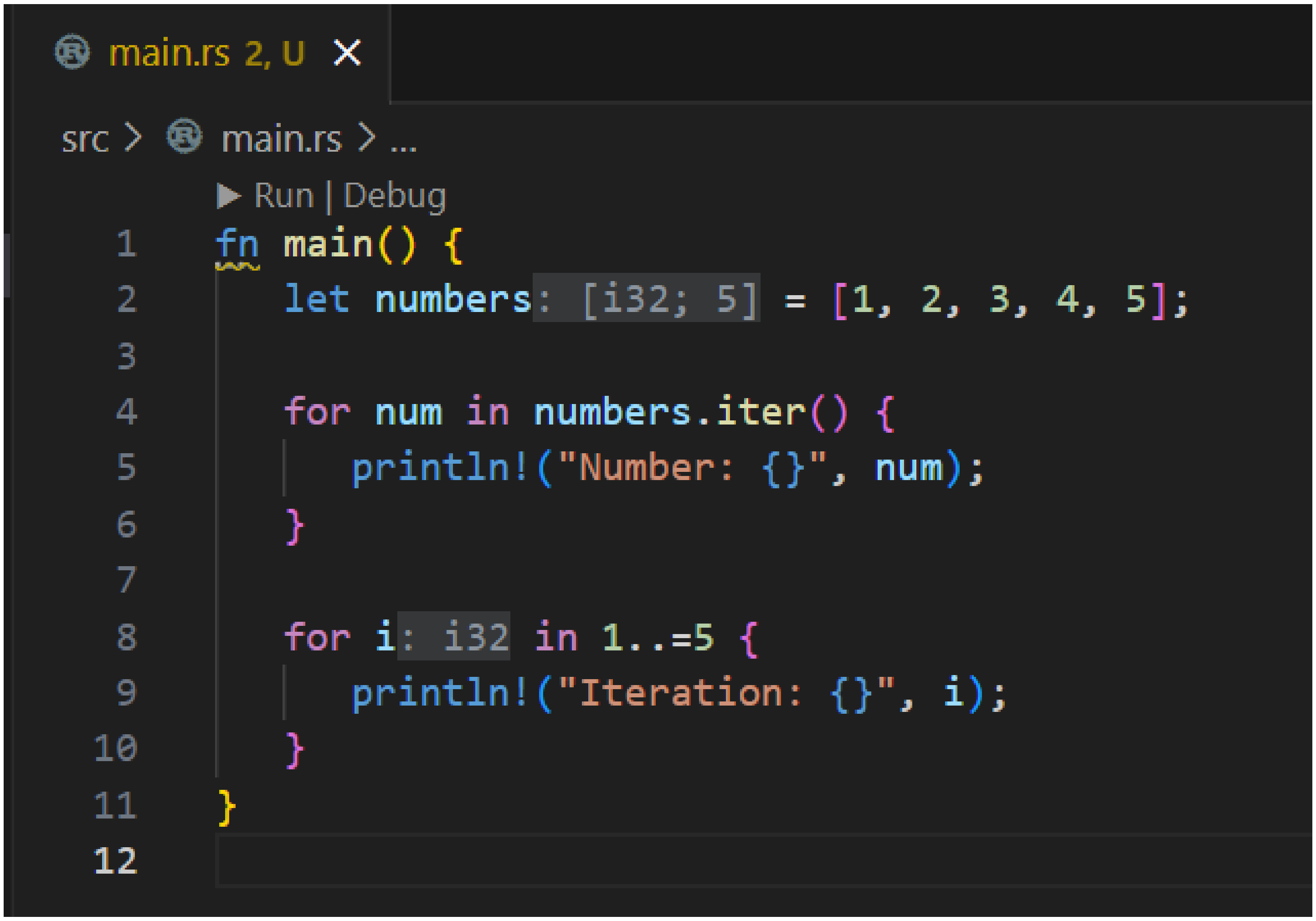


```
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
```

3. For loop: Rust's for loop is used to iterate over a collection of items, such as arrays or ranges. It provides a more convenient and safer way to loop through items compared to a while loop. You can use the for keyword followed by a variable, the in keyword, and the collection you want to iterate over.

For loop

File name: main.rs



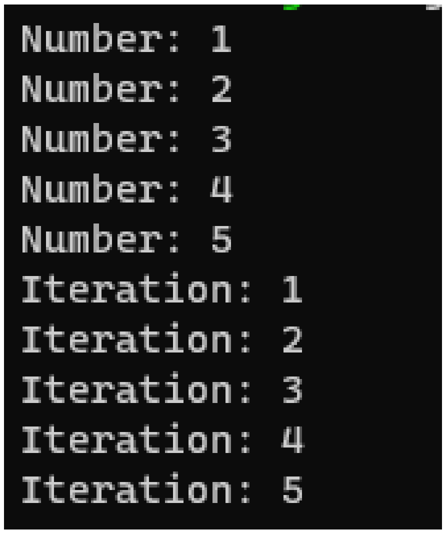
main.rs 2, U

src > main.rs > ...

▶ Run | Debug

```rust
fn main() {
    let numbers: [i32; 5] = [1, 2, 3, 4, 5];

    for num in numbers.iter() {
        println!("Number: {}", num);
    }

    for i: i32 in 1..=5 {
        println!("Iteration: {}", i);
    }
}
```

$ cargo run



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

This is the end of this journey there are advanced Rust concepts and this book only is an intro, hope you find this useful, we focus a lot on the learning experience.
Concepts like: ***borrowing, packages, crates, modules, tests, modules, lists, vectors, Strings, Error Handling, Generic Types, Traits, Tests, Reading Files, I/O***, and much more willbe covered on our next book: "**Intermediate Rust Programming**".
We are working hard to create advanced books in Rust!

Thanks for reading!
Protect Nature.

Rust Programming for Beginners Simplified Version ©