Rust for Beginners

§

Made with Obsidian

Type blog | Category computer-science | Technologies Rust, VS Code, PowerShell 7

Website Post Link

Rust is a compiled, multi-paradigm, low-level, statically-typed, general-purpose programming language emphasizing performance, type safety, and concurrency. It was originally developed as a Mozilla Foundation project in 2013 and has since steadily grown its user base. Despite being a low-level language, it provides safety features that many high-level counterparts also present. Simply put, Rust has better safety standards than other low-level languages (*C, C++*), mainly in the memory management department; Rust has a robust set of safety tools, including null pointer dereferencing prohibition, pointer aliasing enforcement, and most notably, its ownership and borrowing system which we'll talk about later on.

Of course, we're still talking about a low-level language meant to provide direct interaction with the hardware; this means there has to be a way to directly (*and potentially unsafely*) perform low-level operations. This is managed by Rust's unsafe component, also called unsafe Rust, which we'll not thoroughly review in this segment.

We'll be using Rust scripts which can be found in the Blog Article Repo.

§

Table of Contents

§

To Love or Not to Love

According to recent Stack Overflow surveys, Rust has maintained itself as the most loved programming language since 2016; yes, even more loved than Python. It's sometimes thought of as a niche language but has been gaining popularity in recent years.

Its main uses are around systems programming where high performance and safety are required. Some applications include:

- **Game engines**
    - Amethyst
    - Bevy
    - Fyrox
    - Piston
    - Nannou (*Videogames, interactive AV installations, live performance, generative art*)
- **SaaS Applications**
    - Dropbox
    - Discord (*Read States service*)
    - Figma
    - Meta (*Their source control backend*)
    - AWS
- **Infrastructure services**
    - Cloudflare
    - Sentry
    - Firecracker
- **Operating systems**
    - There are discussions of Linux maintainers potentially adopting Rust into the Linux Kernel.
- **CLI-based applications for Unix-like systems**
    - bat
    - ripgrep
    - dust
    - bottom
    - exa
- **Cryptographic key management services**

Of course, Rust is still a relatively new language that has yet to enjoy the massive boom of other languages such as C++, Python, Java, or JavaScript. Still, the adoption rate is increasing, and developers are pushing to grow Rust's crate repository. Some areas, such as ML, are still to be developed further, but the efforts are there and will hopefully continue.

So what is it that makes Rust an attractive option for systems programming? Well, as Alex Gaynor and Geoffrey Thomas at the 2019 Linux Security Summit said:

> Two-thirds of Linux kernel vulnerabilities come from memory safety issues. It's pretty much the same everywhere. Where-ever there's a memory problem, chances are you'll find C or C++.

Let us discuss what makes Rust awesome in more detail.

1. Safe low-level control

Rust provides low-level control by allowing memory management explicitly and efficiently. Rust's ownership and borrowing model offers a safe and efficient way to manage memory, a fundamental aspect of low-level programming. This makes it possible to write secure yet high-performance programs.

In Rust, memory is managed through ownership, meaning each value has a unique owner. When a value is no longer needed, its owner can either move it to a new owner or free the memory it occupies.

Rust also provides borrowing, allowing multiple references to a value to exist simultaneously, but only one can be mutable at a time. This ensures data is accessed safely and prevents data races and other synchronization issues.

2. Concurrency & asynchronous processing

Concurrency refers to the ability of programs to perform multiple tasks simultaneously. Rust provides several tools for concurrent programming, including threads, async/await, and channels, which allow various tasks to be executed concurrently.

This is no special thing since other languages also have this possibility. What makes Rust attractive regarding concurrency is the following:

- Threads in Rust are lightweight and can be created easily using Rust's standard library.
- Due to its robust compiler, many potential concurrent processing problems are pointed out at compile time.
- Rust's concurrency model is based on ownership and borrowing, which provides a safe and efficient way to manage shared data in concurrent programs. This prevents data races and other synchronization issues that may occur in other languages.

Remember Polars? It's implemented in Rust. Polars uses rayon`, a data-parallelism library that allows for easy parallelization of operations on collections without requiring explicit thread management.

3. Backward compatibility

Rust highly emphasizes backward compatibility, particularly between different language versions, libraries, and crates. This is handled by the implementation of two channels:

- A stable release channel.
- A nightly release channel.

The stable release channel is meant for production and receives updates every six weeks. Rust guarantees that code written for one stable release will continue to work on future stable releases as long as it adheres to Rust's language and API specifications.

On the other hand, the nightly release channel is meant for experimental use and receives daily updates. Code written for the nightly release may not be backward-compatible with previous or future versions of Rust, as the language and API specifications may change.

Rust also provides a versioning scheme for libraries and crates, which allows developers to specify which version of a library or crate they are using. This ensures that code written for a given version of a library or crate will continue to work on future versions of that library or crate as long as the library or crate maintains backward compatibility.

This provides several advantages:

- Ease of upgrades
- Reduced maintenance costs
- Increased adoption
- Better user experience

All these are especially important when dealing with a relatively new language that could fundamentally change on each iteration.

4. Debugging

Rust provides two main advantages when it comes to debugging:

- It's not forgivable, at least in the secure Rust implementation; if something shouldn't be there, it will most likely not let us compile our code.
- The compiler error messages are verbose and informative.
- The auto-complete feature in VS Code is simply beautiful. It fills in default data types if we don't specify them and tells us exactly which syntax we should use to do so explicitly.

These two aspects make Rust's debugger useful and a pleasure to work with. It guides us through the writing of our code and tells us when something goes wrong and exactly where the error comes from even before we compile our program.

5. Community & development

Rust is still a fairly new programming language. Still, it has already gained an enthusiastic community of developers known as Rustaceans (*yes, this is a thing*).

As of the writing of this article, the official GitHub Repo has over 4,000 contributors, while the number of Crates in stock is currently at 110,360 (*we'll go over crates further on*), and the number of Crates downloaded surpasses the 30bn mark.

§

What to expect

Although Rust's compiler does a great job of debugging verbose and informatively, it still has a steep learning curve associated, mainly if we come from a high-level, dynamic programming context. However, if we compare Rust with other low-level languages, such as C++, the frustration may not be comparable to that of troubleshooting issues with the latter.

What's hard about Rust is not exclusively the syntax, but systems programming in general, which is what this language was designed to excel at. There is a considerable amount of computer and systems theory behind it. If we want to learn it properly, we must at least have a generalized understanding of concepts such as memory allocation, pointers, borrowing, and data structures.

That said, for those with a background in systems programming, learning Rust will hopefully be a smooth transition, as many of the low-level concepts are similar, if not the same, and some syntactic elements are alike.

In this segment, we will provide an introductory overview of the Rust language. We will also provide enough theory to understand what we're programming, although we will not dive too deep into the Rustacean waters since this is meant to be an introduction to the language.

We'll be using VS Code as well as PowerShell throughout this article.

§

Installation

For this segment, we will need to install three main components:

- The Rust programming language.
- Visual Studio Code.
- The Visual Studio Code Rust extension.

We will also install some packages, which will come later when we get to the package manager.

1. Rust

There are two main methods we can use to install the Rust programming language:

- Via the `rustup` installer.
- Via standalone installers in the form of tarballs (`.tar.gz`) for Unix-like environments, a Windows installer (`.msi`), or a macOS installer (`.pkg`).

We'll stick `rustup`, the recommended method in the official documentation; it has a 6-week rapid release process and supports various platforms. The complete installer documentation can be found here.

To install Rust, we will follow the steps below:

1. Head to the Rust language installation page.
2. Download the 64-bit executable.
3. Run the executable.
4. Take note of the Cargo home directory displayed in the installation prompt (*usually located in* `C:\Users\username\.cargo`).
5. Wait for the following prompt: `You can uninstall at any time...`.
6. Hit `Enter`.
7. Wait for the following prompt: `Rust is installed now. Great!`.
8. Hit `Enter`.
9. Open a new shell and input the following: `rustc --version`
10. If the installed `rustc` version appeared, the installation succeeded.

If, instead, we got an error such as `rustc command not found`, it most probably means that the `.cargo/bin` directory was not added to the `Path` environment variable. To fix this, we will need to add it manually by following the steps below:
1. Open the Windows run command window by typing `WIN`+`r`.

2. Input the following: `"C:\Windows\system32\rundll32.exe" sysdm.cpl,EditEnvironmentVariables`
4. Select `Path`.
5. Click `Edit...`.
6. Click `New`.
7. Input the following: `C:\Users\username\.cargo\bin`.
8. Click `OK`.
9. Click `OK`.
10. Try to execute the `rustc --version` command again.

2. Microsoft C++ Build Tools

To compile Rust code, we will need to install the Microsoft C++ Build Tools. It is probable that we already have this installed. If we don't, we can follow the steps below:

1. Head to the Microsoft Visual Studio Downloads Page.
2. On *All Downloads*, look for `build tools`.
3. Download the *Build Tools for Visual Studio 2022* installer.
4. Install the Microsoft C++ Build Tools.

3. VS Code

If we don't yet have VS Code installed, we can get it from the official downloads page. We need to select the Windows 8, 10, 11 executable and wait for it to download. When the installation is complete, we can verify by opening the Visual Studio Code application directly from the Windows start menu. A detailed configuration guide for VS Code is out of the scope of this article but can be consulted on the VS Code official documentation site.

4. Rust VS Code extension

Once we have Rust and VS Code installed, we will proceed to install the Rust VS Code Extension:

1. Open VS code and head to the Extensions menu in the left panel. We can also open the Extensions menu by using the shortcut `Ctrl` + `Shift` + `X` or by opening the command palette by typing `F1` and searching for *Extensions: Install Extensions*.
2. We will search for `rust-analyzer`, maintained by *rust-lang.org*, and install and enable it. We can also get the extension using this link.

Now that everything's in place, we're ready to start configuring our working environment.

§

Creating a project

The first thing to do after installation is to create a Rust project. This is handled by the `cargo` package manager.

There are two options for creating a new project:

- Creating a new project folder.
- Using an existing folder as the project folder.

If we want to create a project from scratch, we can first head to our project directory:

Code

```
cd Projects
```

We will then initialize our project:

Code

```
cargo new rust-for-beginners
```

Conversely, if we want to use the current directory as the project folder, we can simply change directories to our target folder and initialize it as our project:

Code

```
cd rust-for-beginners
cargo init
```

This command will create two components:

- A `src` folder: Contains our project's source code.
- A `Cargo.toml` file: This is the project's manifest and contains metadata needed to compile our package, such as the project name and required dependencies.

Once we're at it, we can mention that the `.toml` file format is relatively popular among the Rust developer ecosystem, notably used by Cargo, mainly for configuration purposes.

We can optionally install a VS Code extension that will make our life easier when dealing with this file format:

1. Open VS code and head to the Extensions menu in the left panel. We can also open the Extensions menu by using the shortcut `Ctrl` + `Shift` + `X` or by opening the command palette by typing `F1` and searching for *Extensions: Install Extensions*.
2. To install and enable it, we will search for `Even Better TOML`, maintained by *tamasfe*. We can also get the extension by using this link.

This extension is backed by taplo and will support TOML syntax.

1. The project manifesto

If we open our `Cargo.toml` file, we will see that it contains some boilerplate metadata about our newly-created project:

- `package`: Defines a package.
  - `name`: The name of the package.
  - `version`: The version of the package (*not the Rust release*).
  - `edition`: The Rust edition.
- `dependencies`: Package library dependencies.

A `.toml` file general structure consists of sections and key-value pairs, similar to a Python dictionary; a `Cargo.toml` file section groups similar variables together, depending on their functionality.

The complete set of sections and key-value pairs can be consulted here.

Once we start including dependencies in our project, we will also start working on our manifesto. Once the file is generated, the `Cargo.toml` manifesto is meant to be edited by us.

2. Including dependencies

As we mentioned, Rust dependencies are handled by Cargo. There are three main concepts that we should be aware of:

- **Module / Struct:** A module is a way to organize code and data into logical units. A module can contain functions, structs, enums, constants, types, and other modules.
- **Crate:** A crate is a compilation unit in Rust that produces a binary or a library. It contains one or more modules that define functions, structs, enums, traits, and other Rust language constructs. A crate can be compiled into a binary executable or a library that other programs can use.
- **Library:** A library is a special kind of crate intended to be used by other programs. It provides a collection of reusable code that can be linked to other Rust programs. A library can be either a dynamic library (`.so` or `.dll`) or a static library (`.a` or `.lib`). Rust libraries can be organized into modules similar to namespaces in other languages.
- **Package:** A package is a set of one or more crates built and tested together. A package contains a `Cargo.toml` file that specifies the package's name, version, authors, dependencies, build script, and other metadata. A package can be published to the Rust package registry (*crates.io*) to be used by other Rust developers.

Let us head to VS Code and open our `main.rs` source file. Once we open it, we'll notice that two new items have been created in our project folder:

- A `Cargo.lock` file: Contains exact information about our dependencies. It is maintained by Cargo and should not be manually edited.
- A `target` folder: Where the output of a build is stored.
  - A `debug` folder: Where the debug build is stored.
  - A `.rustc_info.json` file: Contains metadata about the compilation process and the target platform for which the code was compiled.
  - A `CACHEDIR.TAG` file: Marks a directory as a cache directory. Tools that recognize the signature inside this file will treat the directory as a cache directory and may take appropriate action, such as excluding it from backups or indexing.

We'll discuss these files & folders in more detail as we move on. For now, we will focus on the `main.rs` file.

We will start by importing the following modules into our main file:

- `std::io`: Standard input/output module.
- `std::cmp::Ordering`: Comparison module.
- `rand::Rng`: Random number generator module.

Code

```rust
use std::io;
use std::cmp::Ordering;
use rand::Rng;
```

If we pay close attention, we can see that we used the keyword `use` to import a library followed by a double colon `::`, the required module(s), nested imports in the case of `std::io` enclosed in curly braces `{}`, and a semicolon `;` termination.

We can also see that two alerts pop up:

- An unused import alert for `std::io`: Rust knows that we have not used the module in our code; hence it raises an alert.
- An unresolved import for `rand`: We have not specified a version for the `rand` library; hence Rust raises an error.

The first alert does not require further action; we will eventually use this module. For the second one, we will head to our `Cargo.toml` manifesto and include the library version under `[dependencies]`:

Code

```toml
[dependencies]
rand = "0.8.5"
```

Once we do this, the error should disappear.

3. The main function

As we will see later, every executable Rust program requires a `main` function. The main function serves as the entry point to our program.

Whenever we create a new Rust project, this main function will be written by default in our `main.rs` file:

Code

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

If we leave this as is, our program will simply print `Hello, world!` to `stdout`.

4. Compiling the project

Compiling and running a Rust project can be done using Cargo and consists of 3 main steps:

- Navigate to the project directory.
- Build the project.
- Run the project.

Code

```
# Navigate to the project directory
cd rust-for-beginners

# Build the project
cargo build

# Run the project
cargo run
```

If the project was built successfully, we'll get the build confirmation as well as the string from our `println!()` statement inside our `main` function:

Output

```
Compiling rust-for-beginners v0.1.0 (C:\Users\username\Documents\rust-for-beginners)
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s

Hello, world!
```

Else, we'll get a build error:

Output

```
Compiling rust-for-beginners v0.1.0
error: Compilation Error Message
```

§

Commenting

Comments are fragments of code that are not included for execution during compilation and running. There are multiple ways to comment in Rust, although we will only review the two most used:

- **Line Comment:** Single line.
- **Block Comment:** Multiline.

1. Line comment

We can comment on a single line by prepending our comment with a double slash `//`:

Code

```
// This is a single line comment.
// Which can also be continued in the next line.
```

2. Block comment

We can comment on a block by enclosing our comment in slash asterisk symbols `/* */`:

Code

```
/*
This is a multiline comment
where we can include multiple lines
without having to comment each line.
*/
```

§

Pointers, references, and ownership

Before diving deeper, it's worth spending some time understanding what pointers are and how Rust's memory, ownership, and references work. This is because these concepts are key in understanding not just Rust's architecture but any low-level language such as C and C++; plus, it will save us a lot of frustration when we get to variables and data types.

1. Pointers

Formally, a pointer is a derived data type that can store the memory address of other variables. It allows us to manipulate and reference data stored in memory by directly accessing its location in memory.

Let us represent a generic block of memory using an array-like structure:

Address	Value
0x0000	01100110
0x0001	00100001
0x0002	10110100
0x0003	11001010

Table 1: Representation Of A Memory Block Using An Array-Like Structure

Where:

- The **address** is the location inside the memory block.
- The **value** is the data stored at that particular address.

In this example, the memory block contains four consecutive locations starting at address 0x0000. Each memory location stores a byte of data represented by a sequence of 8 bits. This example's values are binary, but they can also be represented in hexadecimal or decimal notation.

If we convert our binary sequences to decimal numbers, we get the following:

Address	Value
0x0000	102
0x0001	33
0x0002	180
0x0003	202

Table 2: Binary Characters Converted To Decimal Numbers

If we look closer, the addresses in our memory block are simply numbers represented in hexadecimal form; the `0x` character preceding all the address values denotes we're talking about a hexadecimal value, where the number following would be the actual hexadecimal value.

To convert it to decimal, we simply solve for the following:

$$0x0001 = 0x16^3 + 0x16^2 + 0x16^1 + 1x16^0$$

Simplifying:

$$0x0001 = 0x4096 + 0x256 + 0x16 + 1x1$$

Thus, our decimal number would be 1.

Following this logic, we could define a value in our memory block as an address; this is the definition of a pointer: a value that happens to be an address in memory.

If we substitute the value corresponding to the memory address 0x0002 for another address within the same block and convert the remaining binary representations to hexadecimal 0x notation, we would end up with the following:

Address	Value
0x0000	0x0066
0x0001	0x0021
0x0002	0x0001
0x0003	0x00CA

Table 3: Memory Block Containing A Pointer At Address 0x0002, Referencing Address 0x0001, Or Value 0x0021

Now we have a pointer, but why is this relevant? Well, in Rust, we have three types of pointers:

- References
- Raw pointers
- Smart pointers

We'll focus on the most common type: references.

2. References

In Rust, a reference is a type of pointer that lets us borrow values without taking ownership. This means that we can include an additional layer of abstraction (*e.g., referencing a value without directly interacting with it*), which allows us to borrow values without copying them. Borrowing rules are validated on compile time by the borrow checker.

This can be much more efficient than making copies of large data structures and is especially important in performance-critical code, where reducing memory usage can improve program speed and reduce resource consumption.

As we will see, references in Rust are created using the ampersand `&` operator. For example, if we want to define a reference to the `str` data type and assign it to a variable, we can use the following syntax:

Code

```rust
fn main() {
    let _mystring:&str = "string";
}
```

Not to worry. We'll dive deeper into variable declarations when we get to variables.

3. Ownership

We've already seen an example of borrowing using a data type reference (*i.e., the `str` data type has an owner, and we borrow it from that owner to define a new variable*). This concept is part of a broader idea called ownership.

Ownership in Rust is based on the concept of a unique owner for each value. When a value is created, it is associated with a unique owner responsible for managing the value's memory allocation and lifetime. The owner can be a variable, a function, or a data structure.

Apart from borrowing, we can also transfer ownership using Rust's `move` semantics; when a value is assigned to another owner, the original owner is said to have "*moved*" the value. The new owner takes over responsibility for the value's memory allocation and lifetime, and the original owner becomes invalid.

With this concept, we've also introduced another key idea in Rust: memory safety.

When an owner goes out of scope or is no longer needed, Rust automatically frees the memory associated with the value. This ensures that Rust programs are memory safe, efficient, and free from common low-level programming errors such as dangling pointers, use-after-free, and data races. In short, ownership ensures that there is always exactly one owner for the value and that the owner is responsible for managing the value's memory allocation and lifetime.

Now that we have a general understanding of pointers, references, and ownership, we can discuss variables.

§

Variables

There are several different variable types depending on their structure. Generally, we use `let` to define a variable under the current scope. There are also static variables, although we won't be covering them here.

We must remember that Rust cares about unused variables; if we declare a variable and don't reference it anywhere in our code, the compiler will throw an error unless we prepend our variable name with an underscore `_`. This tells Rust that the defined variable is meant to be used as a placeholder, and thus the compiler will ignore it:

Code

```
fn main() {
    let _myplaceholder:i32 = 34;
    let myplaceholder:i32 = 34;
}
```

The first declaration won't throw an error if left unreferenced, while the second one will.

We can also use underscores `_` in between digits when dealing with numeric variables to denote thousand separators; this will increase legibility while not having any consequence during compilation:

Code

```rust
fn main() {
    // Using underscores as separators
    let mynum_immut = 7_000_000;
    println!("{}", mynum_immut);
}
```

Output

```
7000000
```

As expected, the value will be printed without the underscore characters.

If we recall, we installed a Rust extension for VS Code. This extension provides helpful features such as data type inference upon declaring a variable. We cannot see it directly in our code, but the extension is using the compiler to include the following:

Code

```rust
fn main() {
    // Using underscores as separators
    let mynum_immut: i32 = 7_000_000;
    println!("{}", mynum_immut);
}
```

Note the `i32` specification after the name of our variable.

We can also assign multiple variables in a single statement using tuples (*we'll discuss tuples later in the segment*):

Code

```rust
fn main() {
    // Multiple variable assignment
    let (myvar1, myvar2, myvar3) = (1, 2, 3);
    println!("{} {} {}", myvar1, myvar2, myvar3);
}
```

Output

```
1 2 3
```

1. Immutable variables

Variables in Rust are immutable by default; when a variable is immutable, we can't change a value once a value is bound to a name. This is the default behavior for security purposes; we don't have to keep track of the variables and how they change throughout our program if they cannot be changed in the first place unless we explicitly decide to do so.

The simplest way to define a mutable variable inside a function is as follows:

Code

```rust
fn main() {
    let mystring_immut:&str = "Hello World";
}
```

Where:

- `mystring_immut` is the variable name.
- `&str` is the borrowed form of the `str` type.
- `Hello World` is the value assigned to the variable.

2. Mutable variables

Mutable variables can change throughout our code. The simplest way to define a mutable variable inside a function is as follows:

Code

```rust
fn main() {
    let mut mystring_mut:String = String::new();
}
```

Where:

- `mut` refers to the variable being mutable.
- `mystring_mut` refers to the variable name.
- `String` refers to the data type.
- `new()` tells Rust to create this as a new variable.

3. Scope & blocks

As we have mentioned, variable bindings in Rust have a scope. Scopes are important since they indicate to the compiler when borrows are valid, when resources can be freed, and when variables are created or destroyed.

A block or block expression is a control flow expression and anonymous namespace scope for items and variable declarations.

A block can be defined by using curly brackets:

Code

```rust
fn main() {
    // Defining a block
    let var_outsideblock = 49;
    {
        // Inside a block
        let var_insideblock = 7;
        println!("Inside 1: {}", var_insideblock);
        println!("Inside 2: {}", var_outsideblock);
    }

    // Outside a block
    println!("Outside: {}", var_outsideblock);
}
```

Variables defined inside a block using `let` will not be accessible outside that block. However, variables defined outside a block using `let` will be accessible from within a given block.

This concept also applies to functions, where we can define a variable under the current scope that will not be accessible outside of that function.

§

Constants

Constants in Rust are different from immutable variables in that they are computed at compilation time (*and can be used in other compile-time computation*), and hence the run time is faster, as it does need to compute it again; immutable variables are always computed at run time (*all* `const` *occurrences will be replaced by the value assigned on compilation*). Additionally, immutable variables are defined under the current scope (*are not defined globally*), while constants are global.

Let us elaborate further by using an example:

Code

```rust
// Define constant outside main function
const HELLO: &str = "Constant Hello";

fn main() {
    // Print constant from inside main function
    println!("{}", HELLO);
}
```

As we might have noticed, we define a `const` using uppercase letters.

Output

```
Constant Hello
```

Conversely, if we try to do the same for an immutable variable, the compiler won't even let us declare it outside our function:

Code

```rust
// Define an immutable variable outside main function
let mystring_immut = 7;

fn main() {
    println!("{}", mystring_immut);
}
```

Output

```
consider using `const` or `static` instead of `let` for global variables
```

§

Printing

There are multiple ways we can use to print values in Rust. Below are the two most commonly used:

- `print!()`
- `println!()`

We might have noticed the exclamation mark `!` at the end of each statement because both printing methods are macros (*we'll discuss macros later on*).

Printing a string literal in Rust can be achieved using the `print!()` macro without any additional arguments:

Code

```rust
fn main() {
    // Print a string in same line
    print!("Hello ");
    print!("World");
}
```

Output

```
Hello World
```

This method will not include a newline at the end of each print statement, so it's sometimes best to use the `println!()` macro instead:

Code

```rust
fn main() {
    // Print a string in newline
    println!("Hello");
    println!("World");
}
```

Output

```
Hello
World
```

However, suppose we would like to print other data types different than string literals, such as integer literals, float literals, or variables. In that case, we need to include a string literal format argument using curly brackets `{}`, a placeholder for our variable(s).

Code

```rust
fn main() {
    // Print an integer using a string literal format argument
    println!("{}", 7);
}
```

Output

```
7
```

We can add multiple placeholders, with the arguments to print separated by a comma `,`:

Code

```rust
fn main() {
    // Multiple placeholders
    println!("The numbers are: {} and {}", 7, 4);
}
```

Output

```
The numbers are: 7 and 4
```

We can also add text with the placeholder to format our output:

Code

```
fn main() {
    // Format output with additional text
    println!("The number is: {}", 7);
}
```

If we want to print variables, we can do so by including the same string literal format argument as before:

Code

```
fn main() {
    let intvar = 7421;
    println!("{}", intvar);
}
```

Output

```
7421
```

§

Data types

We mentioned that Rust is a statically-typed language. This means that all data types must be explicitly known to the interpreter before compiling. We have two options when dealing with data types:

- To explicitly define the data type.
- To let the Rust compiler fill in a default data type based on the variable's assigned value.

When working with data types in Rust, there are some core concepts we need to take into account:

- If we have a variable of some type, `T`, we own the data associated with `T`. If we have a variable of type `&T`, then we don't own the data; we're borrowing it.

There are 17 main data types in Rust, which can in turn, be divided into six categories:

- Primitive types
- Sequence types
- User-defined types
- Function types
- Pointer types
- Trait types

Although we will only cover primitive and sequence types in this segment, the full documentation can be consulted here.

1. Primitive types

Primitives refer to the basic data types built into the language and not defined in terms of other data types. There are four main primitive data types in Rust:

- Boolean - `bool`
- Numeric - integer `i` (*signed or unsigned*) and float `f`
- Textual - `char` and `str`
- Never - `!`

1.1 Boolean

The boolean type can take one of two possible values:

- `true`: Has the bit pattern `0x01`
- `false`: Has the bit pattern `0x00`

We can define a simple boolean variable:

Code

```
fn main() {
    let boolean_var:bool = true;
    println!("{}", boolean_var);
}
```

Output

```
true
```

1.1.2 Boolean logic

Boolean values can be operated on using boolean logic or boolean operators. There are five main boolean operators in Rust. Given two variables, `a` and `b`, we can define them as follows:

- Logical not: `!a`
- Logical or: `a | b`
- Logical and: `a & b`
- Logical xor: `a ^ b`
- Logical comparisons:
  - Equal to: `a == b`
  - Greater than: `a > b`
  - Greater than or equal to: `a >= b`
  - Less than: `a < b`
  - Less than or equal to: `a <= b`

1.2 Integer

As mentioned, there are signed and unsigned integer types. The main difference between the two is that:

- A signed integer can hold negative values.
- An unsigned integer can hold a larger positive value and no negative value.

An integer type can also be classified based on its minimum and maximum number of bits (*capacity*), e.g., `i8` (*signed*) will accept a number from -128 to 127, while `i16` (*signed*) will accept a number from -32,768 to 32,767.

This does not just apply to Rust but is common to all programming languages (*it's just that dynamically typed languages handle this automatically for us*) and directly related to the computer hardware and how a system stores information in memory.

Below is the complete list of signed integer data types:

Type	Minimum	Maximum
`i8`	$-(2^{7})$	2^{7}-1
`i16`	$-(2^{15})$	2^{15}-1
`i32`	$-(2^{31})$	2^{31}-1
`i64`	$-(2^{63})$	2^{63}-1
`i128`	$-(2^{127})$	2^{127}-1

TABLE 4: SIGNED INTEGER TYPES WITH THEIR RESPECTIVE MINIMUM AND MAXIMUM VALUES

As well as their unsigned counterparts:

Type	Minimum	Maximum
`u8`	0	2^{8}-1
`u16`	0	2^{16}-1
`u32`	0	2^{32}-1
`u64`	0	2^{64}-1
`u128`	0	2^{128}-1

TABLE 5: UNSIGNED INTEGER TYPES WITH THEIR RESPECTIVE MINIMUM AND MAXIMUM VALUES

It's important to note that the default type for any Rust program will be `i32` (*i.e., if we define an integer numeric variable without explicitly stating its data type, the compiler will most probably suggest* `i32` *as its data type*).

We can define an integer variable under the current scope:

CODE

```
let mynum_i32:i32 = 2000;
```

We can also directly check the maximum number possible for a given data type using the following syntax:

CODE

```
println!("Max size for u8 is: {}", u8::MAX);
```

OUTPUT

```
Max size for u8 is: 255
```

If we try to define a variable with a data type where our value is out of its range, we will get the following error:

Code

```
let mynumber:u8 = 256;
```

Output

```
literal out of range for `u8`
```

1.3 Float

Similar to integers, floating-point values can be classified by bit size and defined by the IEEE 754 standard. The only difference is that there are only two different types we can declare:

- `f32` : "*binary32*" type is 32 bits in size and has single precision.
- `f64` : "*binary64*" type is 64 bits in size and has double precision.

We can define a floating-point variable under the current scope:

Code

```
fn main() {
    let mynum_f32:f32 = 3.1416;
}
```

As with integer types, there is a default floating-point type suggested previously to compilation time; the default type for any Rust program will be `f64` (*i.e., if we define a floating-point numeric variable without explicitly stating its data type, the compiler will most probably suggest* `f64` *as its data type*).

We mentioned precision when discussing the floating-point types available in Rust. This attribute is common to all floating-point types denoting the number of digits after the radix point.

Single precision means a limited set of possible numbers after the radix point, while double precision means more possible numbers.

More concretely, single-precision floating-point numbers provide roughly 6 to 7 significant decimal digits, while double-precision numbers provide approximately 14.

1.4 String

Strings are slightly different in terms of how we use them in Rust; strings and slices are only accessible via references (*borrowing*), thus when we declare a string using the `str` data type, for example, we will have to prepend our data type with an ampersand `&` .

A reference consists of a pointer into memory which cannot be changed directly by the owner of the reference since the owner is not the actual owner of the underlying type.

There are two types of strings in Rust:

- `String`: Is stored as a vector of bytes that can be changed but is always guaranteed to be a valid UTF-8 sequence.

- `&str`: Also called string literal, it's a slice that always points to a valid UTF-8 sequence and can be used to view into a `String`.

To define an empty `String`, we can use the following syntax:

Code

```rust
fn main() {
    let mut mystring1 = String::new();
}
```

We can append to, iterate over, and replace items in a `String`:

Code

```rust
fn main() {
    // Insert a character
    mystring1.push('X');

    // Insert a string
    mystring1.push_str("Y Z");

    // Iterate over words by splitting at whitespaces
    for i in mystring1.split_whitespace() {
      println!("{}", i);
    }

    // Replace a string
    let mystring2: String = mystring1.replace("X", "W X");
    println!("{}", mystring2);


}
```

Note that the first entry we pushed was actually a character `char` and not a string `String`. As we'll see shortly, these two data types are different and, thus, are declared differently:

- String: `""`
- Character: `''`

String indexing can be achieved given that we first convert our `String` to a heap-allocated String:

Code

```rust
fn main() {
    // First convert to heap-allocated string
    let myheapstring: String = mystring2.to_string();
    println!("Heap: {}", myheapstring);

    // Then index
    let mysubstring2 = &myheapstring[1..6];
    println!("Substring: {}", mysubstring2);
}
```

Output

```
Heap: W XY Z
Substring:  XY Z
```

We can also generate a non-empty `String`:

Code

```
fn main() {
    let mystring3 = String::from("This is is a non-empty string");
}
```

So it seems we included a duplicate `is` string in there, so we might want to eliminate it. We mentioned that a `String` type is stored as a vector of bytes that can be changed. Thus, we can first convert our `String` to a vector and then remove duplicate entries:

Code

```
fn main() {
    // First convert to vector
    let mut myvec1: Vec<String> = mystring3.split_whitespace().map(str::to_string).collect();
    println!("{:?}", myvec1);

    // Then remove duplicates
    myvec1.dedup();
    println!("{:?}", myvec1);
}
```

Output

```
["This", "is", "is", "a", "non-empty", "string"]
["This", "is", "a", "non-empty", "string"]
```

Once we have a vector, it's easy to perform various operations & transformations, as we'll see when we get to vectors.

1.5 Character

A character `char` is similar to a string, the main difference being the first is always four bytes, while the latter doesn't have to be composed of just four-byte chunks.

We can declare a char under the current scope by using single quotes `''`:

Code

```
fn main() {
    let mychar:char = 'a';
}
```

If we try to include more than one character, we will get an error:

Code

```
fn main() {
    let mychar:char = 'ab';
}
```

Output

```
character literal may only contain one codepoint
Syntax Error: Literal must be one character long
```

1.6 Never

The never type `!` is a type with no values, representing the result of incomplete computations. As of Rust 1.68.2 (*current version as of April 2023*), the `!` type can only appear in function return types. Expressions of type `!` can be coerced into any other type.

2. Sequence types

There are three main sequence data types in Rust:

- Tuple - `( Type_1 , Type_2, Type_n )`
- Array - `[ Type ; Expression ]`
- Slice - `[ Type ]`

2.1 Tuple

A tuple can have members of different types but cannot change size. Different data types mean we must define each item's types inside the tuple. Also, tuples are immutable by default, but we can define a mutable tuple using the `mut` keyword.

2.1.1 Immutable tuples

A tuple is defined using the following syntax:

Code

```
fn main() {
    let mytuple: (i32, i32, &str, f32, bool) = (12, 14, "16", 18.0, true);
}
```

We can index the contents of a tuple:

Code

```rust
fn main() {
    let mytuple: (i32, i32, &str, f32, bool) = (12, 14, "16", 18.0, true);

    println!("{}", mytuple.0)
}
```

Output

```
12
```

We may have noticed that the indexing method differs from other languages. We don't use the typical square bracket notation `[]` to index a tuple. Instead, we use the dot notation `.`.

2.1.1 Mutable tuples

A mutable tuple is defined using the `mut` keyword:

Code

```rust
fn main() {
    // Define a mutable tuple
    let mut mytuple_mut: (i32, i32, &str, f32, bool) = (12, 14, "16", 18.0, true);

    println!("Original: {:?}", mytuple_mut);

    // Mutate the tuple
    mytuple_mut.0 = 14;

    println!("Mutated: {:?}", mytuple_mut);
}
```

Output

```
Original: (12, 14, "16", 18.0, true)
Mutated: (14, 14, "16", 18.0, true)
```

Although tuples have multiple use cases, they also have limitations due to their nature (*relation with structs*):

- Tuples cannot be used as iterable sets.
- We cannot return the length of a tuple directly (*we can write a function to do so, but it gets messy and is not performance-friendly*).
- Tuples can be printed using the same format method as with arrays (*as we'll see in a moment, they require different formatting*).
- Long Tuples (*more than 12 elements*) cannot be printed.
- Tuples have fixed length - that is, the number of values they can store is fixed. We call this "*arity*".
- The elements of a tuple have no name, although we can access its values using an index.

2.2 Array

An array is strictly implemented; it has to be defined at compile time (*using literals*), be of a single data type, cannot change in size, and is immutable by default.

2.2.1 Immutable arrays

An immutable array can be defined using squared brackets `[]`:

Code

```rust
fn main() {
    let myarr:[i32; 10] = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100];
}
```

If we recall, Rust is statically typed; thus, we have to define the data type of all the variables we write. Additionally, arrays cannot change in size, and their size must be known to the compiler on compile time, so we include this size as part of the variable declaration.

We can perform multiple operations on an array:

Code

```rust
fn main() {
    // Index the first element
    println!("{}", myarr[0]);

    // Get its length
    println!("{}", myarr.len());

    // Multiply the first element by a scalar
    println!("{}", myarr[0] * 10);
}
```

Output

```
10
10
100
```

We must remember that Rust uses zero-based numbering, meaning the first element of any sequential data structure will always be `0`.

Printing arrays is slightly different since the `println!()` macro with the simple `{}` formatter cannot handle array structures. If we want to print an array using this macro, we need to change the print formatting to include `:?` inside the curly brackets `{}`:

Code

```rust
fn main() {
    let myarr:[i32; 10] = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100];
    println!("{:?}", myarr);
}
```

Output

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

2.2.1 Mutable arrays

We can also define a mutable array by prepending the `mut` keyword to the variable name:

Code

```rust
fn main() {
    // Define a mutable array
    let mut myarr_mut:[i32; 10] = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100];

    // Print the original array
    println!("Original: {:?}", myarr_mut);

    // Substitute a value
    myarr_mut[0] = 1000;

    // Print the mutated array
    println!("Mutated: {:?}", myarr_mut);
}
```

Output

```
Original: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Mutated: [1000, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

We can also iterate over an array, although we'll review this once we get to the *Iterators* section later.

2.3 Slice

A slice is a dynamically sized type representing a "*view*" into a sequence of type `T` elements. The slice type is written as `[T]`.

Slices are useful for allowing safe, efficient access to a portion of an array without copying. For example, we might want to reference just one line of a file read into memory. By nature, a slice is not created directly but from an existing variable. Slices have a length, can be mutable or not, and in many ways, behave like arrays.

Some moments ago, we used slices to extract a substring from a `String`. We can do the same for other data types as well:

Code

```
fn main() {
    // Declare mutable array
    let mut myarr2: [i32; 5] = [1, 2, 3, 4, 5];

    // Declare mutable slice
    let myslice5 = &mut myarr2[1..4];

    // Mutate slice
    myslice5[0] = 100;
    println!("Slice 5: {:?}", myslice5);
}
```

Output

```
Slice 1: [2, 3, 4]
Slice 2: [1, 2, 3, 4]
Slice 3: [2, 3, 4, 5]
Slice 4: [1, 2, 3, 4, 5]
Slice 5: [100, 3, 4]
```

The source type must also be mutable for us to declare a mutable slice from another type.

3. Casting data types

We can convert to different data types using explicit type conversion, called casting. If we try to convert a type implicitly, the compiler will return an error.

A casting in Rust is achieved using the `as` keyword:

Code

```
fn main() {
    let myfloat: f32 = 32.7;
    println!("{}", myfloat);
    let myint: i32 = myfloat as i32;
    println!("{}", myint);
}
```

Output

```
32.7
32
```

Note that when casting a floating-point number to an `int` type, Rust will truncate the number up to the decimal part, meaning it will not get rounded up or down; it will simply remove the latter.

§

Other types

We already discussed three sequence data types: `tuple`, `array`, and `slice`.

Data structures or collections:

- Vector
- Hash map

1. Vector

Vectors in Rust are re-sizable arrays (*if defined as mutable*). What makes them attractive vs. other data types is that they have associated several methods we can use to interact with them. In fact, we just saw how a collection of strings or characters could be converted to a vector, opening the possibility to sort, remove duplicates, and perform all sorts of interesting operations.

The downside to vectors is that they accept values with a single data type.

We can define an empty vector or a populated immutable vector using the following syntax:

Code

```rust
fn main() {
    // Define an empty vector
    let myvector1: Vec<i32> = Vec::new();
    println!("{:?}", myvector1);

    // Define a populated mutable vector
    let myvector2: Vec<i32> = vec![10, 20, 30, 40];
    println!("{:?}", myvector2);
}
```

Output

```
[]
[10, 20, 30, 40]
```

We might have noticed that we appended an exclamation mark at the end of the `vec` statement. As we have seen with `println!()`, we're using a macro.

We can also define a mutable vector and perform some operations on it:

Code

```
fn main() {
    // Define a populated mutable vector and perform some operations
    let mut myvector3: Vec<i32> = vec![50, 60, 70, 80];

    // Append number
    myvector3.push(90);

    // Index first component
    println!("{}", myvector3[0]);

    // Verify if a given value exists
    match myvector3.get(1) {
      Some(60) => println!("Number 60 is in the vector."),
      _ => println!("Number 60 is not in the vector."),
    };

    // Print length of vector
    println!("{:?}", myvector3.len());
}
```

Output

```
50
Number 60 is in the vector.
4
```

If we had not defined the `push()` method after declaring our mutable vector, the compiler would've thrown an error. This is because Rust checks if a mutable set was, in fact, mutated at some point in the code. If this is not the case, Rust requires us to roll back to an immutable object. This feature is part of Rust's security measures and becomes useful when designing complex programs.

We can perform more operations on vectors, although we'll not cover them all in this segment. The complete documentation can be consulted here.

We will also not include hash maps. Still, they can be consulted here.

§

Operators

Rust has a collection of default logical, arithmetic, and method operators. Each operator is available to a data type or set of data types based on their nature.

Below are the most commonly used operators, although the full set can be consulted here.

Operator	Example	Explanation
`!`	`ident!(...), ident!{...}, ident![...]`	Macro expansion
`!=`	`expr != expr`	Nonequality comparison
`%`	`expr % expr`	Arithmetic remainder
`%=`	`var %= expr`	Arithmetic remainder and assignment
`&`	`&expr, &mut expr`	Borrow
`&`	`expr & expr`	Bitwise AND
`&=`	`var &= expr`	Bitwise AND and assignment
`&&`	`expr && expr`	Short-circuiting logical AND
`*`	`expr * expr`	Arithmetic multiplication
`*=`	`var *= expr`	Arithmetic multiplication and assignment
`*`	`*expr`	Dereference
`+`	`expr + expr`	Arithmetic addition
`+=`	`var += expr`	Arithmetic addition and assignment
`,`	`expr, expr`	Argument and element separator
`-`	`- expr`	Arithmetic negation
`-`	`expr - expr`	Arithmetic subtraction
`-=`	`var -= expr`	Arithmetic subtraction and assignment
`.`	`expr.ident`	Member access
`..`	`.., expr.., ..expr, expr..expr`	Right-exclusive range literal
`..=`	`..=expr, expr..=expr`	Right-inclusive range literal
`/`	`expr / expr`	Arithmetic division
`/=`	`var /= expr`	Arithmetic division and assignment
`:`	`pat: type, ident: type`	Constraints
`;`	`expr;`	Statement and item terminator
`<<`	`expr << expr`	Left-shift
`<<=`	`var <<= expr`	Left-shift and assignment
`<`	`expr < expr`	Less than comparison
`<=`	`expr <= expr`	Less than or equal to comparison
`=`	`var = expr, ident = type`	Assignment/equivalence
`==`	`expr == expr`	Equality comparison
`=>`	`pat => expr`	Part of match arm syntax
`>`	`expr > expr`	Greater than comparison
`>=`	`expr >= expr`	Greater than or equal to comparison
`>>`	`expr >> expr`	Right-shift
`>>=`	`var >>= expr`	Right-shift and assignment

Operator	Example	Explanation
`^`	`expr ^ expr`	Bitwise exclusive OR
`^=`	`var ^= expr`	Bitwise exclusive OR and assignment
`|`	`pat | pat`	Pattern alternatives
`|`	`expr | expr`	Bitwise OR
`|=`	`var |= expr`	Bitwise OR and assignment
`||`	`expr || expr`	Short-circuiting logical OR

Table 6: Most Common Operators In Rust

For example, we can use arithmetic operators:

Code

```rust
fn main() {
    let mut myvar1: i32 = 14;
    let mut myvar2: i32 = 16;

    // Arithmetic operations
    println!("Addition: {}", myvar1 + myvar2);
    println!("Subtraction: {}", myvar1 - myvar2);
    println!("Product: {}", myvar1 * myvar2);
    println!("Division: {}", myvar1 / myvar2);

    // Arithmetic operations with assignment
    myvar1 += 1;
    myvar2 /= 2;
    println!("Addition with assignment: {}", myvar1);
    println!("Division with assignment: {}", myvar2);
}
```

Output

```
Addition: 30
Subtraction: -2
Product: 224
Division: 0
Addition with assignment: 15
Division with assignment: 8
```

1. Alternative methods for common operators

Some mathematical operations which would be directly defined in other languages as operators, are not available in Rust. Two common examples are the power and the modulus operator.

1.1 Power operator

We can elevate any number to the n^{th} power by using the `.pow` method:

Code

```
fn main() {
    let base:i32 = 7;
    let power: u32 = 2;
    let elevated:i32 = base.pow(power);
    println!("{}", elevated);
}
```

Output

```
49
```

The power exponent must be of type `u`, which means it can only contain a positive value. If we try to define our exponent as a signed integer `i`, we will get an error in return.

§

Conditional control

As with many other programming languages, Rust provides a couple of ways to control the flow of our program using conditional statements.

1. If, else if, else

We can define a simple `if`, `else if`, `else` statement using double ampersand operators `&&` to logically test two conditions using `and`:

Code

```
fn main() {
    let firstnum:i32 = 7;
    let secondnum:i32 = 15;
    let thirdnum:i32 = 14;

    if (secondnum > firstnum) && (secondnum > thirdnum) {
      println!("Number {} is the biggest one of the series.", secondnum);
    } else if (firstnum > secondnum) && (firstnum > thirdnum) {
      println!("Number {} is the biggest one of the series.", firstnum);
    } else {
      println!("Number {} is the biggest one of the series.", thirdnum);
    }
}
```

Output

```
Number 15 is the biggest one of the series.
```

We can do something similar with double pipes `||` to test for `or`:

Code

```
fn main() {
    if (firstnum % 2 == 0) || (secondnum % 2 == 0) || (thirdnum % 2 == 0) {
      println!("There is at least one even number here.");
    }
}
```

Output

```
There is at least one even number here.
```

We can also test a statement by assigning our condition to a variable. This is relevant when we have conditions that are confusing or large, and we would like to provide more clarity to the reader:

Code

```
fn main() {
    let firstnum:i32 = 7;
    let fourthnum:i32 = 49;
    let base: u32 = 2;
    let numbertest:bool = if firstnum.pow(base) == fourthnum {true} else {false};
    println!("{}", numbertest);
}
```

Output

```
true
```

2. Match

Rust provides pattern matching via the `match` keyword, which can be used similarly to a C `switch` or a bash `case` statement. `match` is not a method nor a library, but is built into the Rust language itself.

We can think of a `match` implementation as a collection of cases where all the cases are compared against a given value.

2.1 A simple case

The generic syntax for `match` is as follows:

Code

```
fn main() {
    let mynum:u32 = 14;
    let mynum_reminder:u32 = mynum%2;
    match mynum_reminder {
      0 => println!("The number {} is even.", mynum),
      1 => println!("The number {} is odd.", mynum),
      _ => println!("{} is not a number.", mynum),
    };
}
```

Output

```
The number 14 is even.
```

We might have noticed some interesting details:

- We use a fat arrow symbol `=>` to point to the result if a given case is true.
- We use a comma `,` to separate each case.
- We use curly brackets `{}` to enclose the set of cases to evaluate.
- We close our match statement by using a semicolon `;`.
- We include an underscore sign `_` as the last condition. This tells `match` to include all other possible cases apart from `0` and `1`; even though by common sense, we only have two possible options for the reminder calculation (`0`, `1`) as its type is unsigned (*i.e., we cannot have negative values*), we could ask the user to input a value for `mynum`, opening the possibility for a `None` value. If we wanted to be more thorough with a more complex set of cases, we could also include `MIN` and `MAX` as boundaries.

Related to the point above, it's important to mention that when working with `match` statements, the options must be exhaustive, meaning we cannot let room for result ambiguity. If we do, the compiler will return an error:

Code

```
fn main() {
    let mynum:u32 = 14;
    let mynum_reminder:u32 = mynum%2;
    match mynum_reminder {
      0 => println!("The number {} is even.", mynum),
      1 => println!("The number {} is odd.", mynum),
    };
}
```

Output

```
Non-exhaustive patterns: `2_u32..=u32::MAX` not covered.
Ensure that all possible cases are being handled by adding a match arm with a wildcard pattern or
an explicit pattern as shown: `,
```

2.2 Using compare and ordering

Earlier in this segment, we imported the comparison library, including the `Ordering` method `std::cmp::Ordering`. We can use the `cmp()` method to make direct comparisons between variables using a simpler syntax:

Code

```
fn main() {
    let mynum1:u32 = 14;
    let mynum2:u32 = 15;
    match mynum1.cmp(&mynum2) {
      Ordering::Less => println!("{} is less than {}.", mynum1, mynum2),
      Ordering::Greater => println!("{} is greater than {}.", mynum1, mynum2),
      Ordering::Equal => println!("{} is equal to {}.", mynum1, mynum2),
    };
}
```

Output

```
14 is less than 15.
```

We have just reviewed a couple of `match` implementations, but this is just a tiny sample; it includes a variety of methods that can be used to make more complex comparisons and even destructure items such as tuples, arrays, slices, and even pointers in a variety of ways. The complete documentation can be consulted here.

§

Random

At the beginning of this segment, we imported the `rand::Rng` random number generator module. Rust has multiple ways to generate random numbers, though we'll only discuss a couple.

The first method consists of using the `rand::thread_rng.gen_range()` random number generator:

Code

```
use rand:Rng;

fn main() {
    let myrandnum = rand::thread_rng().gen_range(1..11);
}
```

Where:

- `myrandnum` will be a random number generator.
- `gen_range(1..11)` will denote a generator taking values from 1 up to 10 (*inclusive*).

The value corresponding to the `myrandnum` variable will change randomly after each assignment.

§

Loops & iterators

Loops & iterators in Rust are patterns that allow sequential access to a collection of values, one at a time. They are similar in concept to loops and iterators from other languages, such as `for` and `while` loops and the `iter()` function in Python, but are defined differently.

There are four main ways we can use to iterate over a set of values:

- Using `loop`
- Using `while`
- Using `for`
- Using iterators `iter()` or `range`

1. Using loop

We can loop through a set of values using the `loop` keyword. A `loop` iterates infinitely many times until a `break` statement is encountered; it will keep looping through the block we define as the body of our `loop` statement until a `break` statement is encountered or an error is returned:

Code

```
fn main() {
    // Define loop (without counter condition check)
    loop {
      println!("{}", myarray[mycounter]);

      // Increase counter
      mycounter += 1;
    }
}
```

Output

```
1
2
3
4
5
thread 'main' panicked at 'index out of bounds: the len is 5 but the index is 5'
error: process didn't exit successfully: `target\debug\rust-for-beginners.exe` (exit code: 101)
```

If we look closely at the output, the program did run, but we got an error in the end; the `loop` did not stop until our counter index was out of the array's range.

If we want to stop the `loop` before the index goes out of range, we must explicitly specify a conditional statement:

Code

```
fn main() {
    // Define an array
    let myarray:[i32; 5] = [1,2,3,4,5];

    // Define a counter as a mutable variable
    let mut mycounter:usize = 0;

    // Define loop (without counter condition check)
    loop {
      // First, check condition. If it's true, break
      if mycounter >= myarray.len() {
      break;
      }

      // If condition is false, execute code
      println!("{}", myarray[mycounter]);

      // Increase counter
      mycounter += 1;
    }
}
```

Output

```
1
2
3
4
5
```

This method is unproductive since we must manually check each iteration's condition. There are more productive methods to loop over sets.

2. Using while

A slightly more familiar method would be a `while` loop. Like other languages, it can run a loop while a given condition is true.

Following the same example as above, we can define a `while` loop:

Code

```rust
fn main() {
    // Define an array
    let myarray:[i32; 5] = [1,2,3,4,5];

    // Define a counter as a mutable variable
    let mut mycounter:usize = 0;

    // Define loop
    while mycounter < myarray.len() {

      println!("{}", myarray[mycounter]);

      // Increase counter
      mycounter += 1;
    }
}
```

In contrast to the `loop` we defined earlier, here we must set our condition as `mycounter` to be strictly less `<` than the length of our array. If we were to set it as less than or equal to `<=`, whenever the loop reaches `myarray = 5`, the index is out of range, and the program returns an error.

3. Using for and range

A for `loop` is also widely used in other programming languages. The main difference is that in Rust, we must use a range along with it if we don't have an iterable set in hand.

The basic construct consists of the following:

Code

```rust
fn main() {
    // Define an array
    let myarray:[i32; 5] = [1,2,3,4,5];

    // Define loop
    for i in 0..myarray.len() {
      println!("{}", myarray[i]);
    }
}
```

Output

```
1
2
3
4
5
```

We can already see that the syntax is cleaner than a `loop` or even a `while` loop. There are a couple of details we must remember, though:

- We define a `range` by using the `start..end` notation.

- The range is denoted as `[start, end)`, where `end` is not inclusive, while `start` is.
- The index `i` may or may not be referenced inside the loop (*i.e., we know that in Rust, we get errors if we leave a variable unreferenced. An index in a `for` loop does not require referencing*)

4. Using for and iter

We can use iterator sets to iterate directly over them. To loop through a collection of values such as an array, we first have to define an iterable set, for example, by using a previously-defined array:

Code

```rust
fn main() {
    // Define an array
    let myarray:[i32; 5] = [1,2,3,4,5];

    // Define an iterable set
    let myiter = myarray.iter();

    // Define loop
    for i in myiter {
      println!("{}", i)
    }
}
```

Output

```
1
2
3
4
5
```

This is by far the cleanest method in terms of syntaxis since we can even define our iterator after the for loop declaration:

Code

```rust
fn main() {
    // Define an array
    let myarray:[i32; 5] = [1,2,3,4,5];

    // Define loop including iterable set
    for i in myarray.iter() {
      println!("{}", i)
    }
}
```

Output

```
1
2
3
4
5
```

5. Using for with enumerate

Last but not least, we can use a `for` loop with the `enumerate()` method.

The enumerate method will return an index value pair per iteration, both of which we will assign to two separate variables:

Code

```rust
fn main() {
    // Define loop including iterable set and enumerate
    for (index, item) in myarray.iter().enumerate() {
      println!("{}: {}", index, item);
    }
}
```

Output

```
0: 1
1: 2
2: 3
3: 4
4: 5
```

6. Collecting an iterator

If we define an iterator using a `range`, we can transform it back to another data structure, such as a vector or an array.

We do this by using the `collect()` method:

Code

```rust
fn main() {
    // Define a range iterator
    let myrange = 1..12;

    // Collect iterator to vector
    let mycollectedrange: Vec<i32> = myrange.collect();
    println!("{:?}", mycollectedrange);
}
```

Output

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

§

Functions

Functions in Rust can be declared by using the following syntax:

Code

```
fn myfun(args) {
    function_body;
}
```

Where:

- `args` represents the arguments we will pass to our function.
- `function_body` represents our function's content, terminated by a semicolon `;` (*content should be indented using 4 spaces*).

It's important to note that, unlike other languages, all code in Rust, except counted cases such as constants, must be written inside a function or a module. Similarly, functions must be called from within other functions or modules.

5.1 Main function

In Rust, the `main` function is used to signal the start of program execution and control flow throughout the program. It does not accept any arguments and should be included (*although it's not required*) as the first function in our program. Until now, we've stuck to using the `main` function exclusively; it gets executed directly upon compilation and run of our application.

A typical main function syntax is shown below:

Code

```
fn main() {
    function_body;
}
```

As we discussed, we do not have to explicitly call our `main` function since it gets executed directly. It makes sense, then, to include all our other function calls inside the main function.

5.2 User-defined functions

Apart from the `main` function, we can define our own using the following syntax:

Code

```
fn myfun1() {
    let myint: i32 = 100;
    println!("{}", myint);
}
```

Since we have not yet defined another function other than `main`, we need to call `myfun1` from within `main`:

Code

```
fn main() {
    myfun1();
}
```

Output

```
100
```

5.3 Functions with arguments

As usual, if we want to declare a function that accepts arguments, we must declare each argument's type in the function definition:

Code

```
fn myfun2(myarg1: i32, myarg2: f32) {
    println!("{}, {}", myarg1, myarg2);
}

fn main() {
    myfun2(21, 22.3);
}
```

Output

```
21, 22.3
```

5.4 Functions returning a value

If we want to define a function that returns a value to the user, we can use two different methods:

- Without explicitly including a `return` statement.
- Explicitly including a `return` statement.

The first one is useful when we're simply performing one operation, while the latter is clearer when reading:

Code

```rust
fn myfun3(myarg1: i32, myarg2: i32) -> i32 {
    myarg1 * myarg2
}

fn myfun4(myarg1: i32, myarg2: i32) -> i32 {
    let op1: i32 = myarg1 - 2;
    let op2: i32 = myarg2 - 4;
    let op3: i32 = op1 * op2;

    return op3;
}

fn main() {
    // Assign output of myfun3
    let myoutput3: i32 = myfun3(7, 7);
    println!("{}", myoutput3);

    // Assign output of myfun4
    let myoutput4: i32 = myfun4(7, 7);
    println!("{}", myoutput4);
}
```

Output

```
49
15
```

We can also return multiple values as outputs using a tuple `()`; the assignment in the main function would also be handled with a tuple `()`.

§

Macros

Macros in Rust are a way of writing code that writes other code. In simpler terms, a macro expands to produce more code than the code we've written manually. All macros are terminated using an exclamation mark `!`.

A simple macro example is `println!(arg)`, which prints `arg` in a newline.

Although we'll not be discussing macros in detail, the full set of Rust macros can be consulted here.

§

User input

We can accept user inputs using the `std::io` module. The simplest way to define user input is as follows:

Code

```rust
use std::io;

fn main() {
    // Mutable variable
    let mut mystring_mut:String = String::new();

    // Accept user input
    io::stdin().read_line(&mut mystring_mut)
    .expect("Didn't Receive Input");
}
```

Where:

- `read_line` is a method belonging to the `std::io::BufRead` method, which reads a line of input, appending it to the specified buffer.
- `&mut` denotes a mutable reference to our previously declared binding variable.
- `mystring_mut` is the binding variable name.
- `expect`: The `read_line` method will return a result of type `enum` (*provide a way of saying a value is one of a possible set of values*), where the result will be either `Ok` or `Err`. If the input reading fails, the `read_line` method will return `Err`, along with the error message inside the `expect` method.

If the return value is `Ok`, the user input will be stored in our variable `mystring_mut`, which we can use in our code as with any other variable of the same type.

§

Next steps

We covered just a fraction of what Rust can do. This language has endless potential for systems programming, web development, cryptographic applications, cross-platform development, CLI-based application development, and many more exciting applications.

Regardless of what we're trying to achieve, the most probable next step to get more familiar with the language is to consult the official documentation. Rust provides three ways to do so:

- **The Rust Book**: The first stop for most aspiring Rustaceans, this book is available as an online index or as an e-book / paperback published by No Starch Press.
- **The Interactive Rust book**: A cool experiment by Brown University aimed at transforming the Rust book into an interactive environment containing quizzes, highlighting, visualizations, and many more features.
- **Rust by example**: An extremely user-friendly collection of runnable examples that illustrate various Rust concepts and standard libraries.
- **Rustlings (highly recommended)**: A collection of small Rust exercises covering all fundamental topics.

There are also a number of unofficial free resources online:

- **Rust Tutorial Full Course**: This segment was heavily inspired by Derek Banas's material. Rust Tutorial Full Course is a great introductory video covering the key aspects of the language.
- **Rust 101 Crash Course**: A great 6-hr marathon including a comprehensive overview of Rust's most relevant features as well as practice exercises.

Additionally, there are some additional books to complement the learning process:

- **Programming Rust, 2nd Edition by O'Reilly** is hands-down the best complementary book to consult.
- **Rust Programming Cookbook**: Yet another fully packed book containing all the nits and grits of Rust.
- **Rust in Action**: A great resource whose cover resembles that of Charles Mulligan's Steakhouse.

I would also heavily advise visiting three key places:

- **The Rust Community**: Where Rustaceans talk about Rust.
- **The Official Rust Repo**: Where Rust and issues live (*useful when trying to debug*).
- **creates.io:** Where Rust crates are hosted. Taking a tour around crates.io could shed additional light on Rust's capabilities.

In terms of what could be learned next, I would recommend tackling the following topics, all of which can be consulted in the official documentation:

- **Reinforcement of key concepts:**
  - Borrowing
  - Variable shadowing
  - Heap vs. stack memory
  - Concurrency
  - Threads & multi-threading
  - Traits
- **Other key types:**
  - Box
  - Hash Maps
  - Structs
  - Other pointer types
- **Further modularization:**
  - Closures
  - Modules
- **User & file interaction:**
  - File I/O
  - File manipulation
  - More elaborate handling if user inputs
  - Buffer

And last but not least, the best advice anyone will ever give: stop reading and write some code.

§

Conclusions

In this segment, we installed the Rust programming language from scratch along with Visual Studio as an IDE and the Visual Studio Rust extension.

We learned how to create Rust projects using cargo, install packages, manage them appropriately using our project's manifesto, and write our first Rust program.

We also explained some of the theory behind the Rust language, including its main advantages, as well as core concepts regarding low-level programming, such as memory allocation, pointers, references, and ownership.

Finally, we provided some next steps for those interested in becoming Rustaceans.

Rust is extensive and has a variety of features available that make systems programming easier and safer. It has a vast community of enthusiasts and experts constantly advising on subjects ranging from full implementations to syntactical details to complex multi-threading systems. Additionally, the official learning resources are impressive; interactive books and problem sets provide an entertaining medium to practice Rust.

This is, without a doubt, a sophisticated language based on a sophisticated set of rules designed to make systems programming safer and more efficient. We only hope it will eventually get the adoption it deserves.

§

References

- Peter Wayner, 7 reasons to love the Rust language—and 7 reasons not to
- Rust Lang, Safe & Unsafe Rust
- Codilime, Rust vs. C++
- Techrepublic, The most loved and most disliked programming languages revealed in Stack Overflow survey
- Derek Banas, Rust Tutorial Full Course
- Rust Lang, Rustup
- Rust Lang, The Manifest Format
- Rust Lang, Cargo Toml vs. Cargo Lock
- Rust Lang, Types
- LogRocket, Rust Iterators
- Rust Lang, Operators
- Brandeis University, Arrays, Vectors, and Slices in Rust

§

Copyright