Rust Mini Reference 2023

A Quick Guide to the Rust Programming Language

Harry Yoon

Version 1.2.3, 2023-04-28

Copyright

Rust Mini Reference:
A Quick Guide to the Rust Programming Language

Published: April 2023

Harry Yoon
San Diego, California

ISBN: 9798392452019

Preface

The legend has it that Rust was born from C++ father and Haskell mother.

Well, we just made that up. ☺ But, once you start using Rust, you will most likely agree with that statement. Rust was undoubtedly influenced by many different programming languages including imperative languages like C#, Go, and Python and functional languages like Lisp, F#, and Scala, among many others. But, Rust is ultimately a hybrid language built from C/C++ and OCaml/Haskell.

If you are experienced with one or the other lineage, or both, then you will feel right at home with Rust. It will be pretty easy for you to learn and use. On the flip side, if you don't have much experience with either side, then Rust will be a rather difficult language for you. You should ask yourself why you want to learn and use Rust before embarking on this (potentially long) journey of "oxidation". To put it bluntly, Rust is not for everyone.

Rust is a paradox. Although it was originally created to be a low-level systems programming language, it includes many high-level constructs, largely influenced by functional programming languages.

At the risk of oversimplification, a programming language can be classified into one of two categories. There are languages, on the one hand, that handle memory management on behalf of the programmers, for example, by providing some kind of garbage collection implementations. These languages are generally considered more high level.

Those in the other category, on the other hand, give as much control to the programmers as possible, which generally makes them more low level. In C, for instance, programmers need to take care of all memory allocations and de-allocations. C++ provides slightly more high level constructs like constructors and destructors. But, in the end, it is the responsibility of the programmer to manage the memory use of their programs.

Rust, interestingly, lies somewhere between these two categories. In a sense, Rust belongs to the second category. There is no garbage collection runtime built into Rust. Programmers should take care of memory management, e.g., by strictly following certain rules imposed by Rust. In another sense, Rust belongs to the first high-level language category. As long as the programmers follow these rules imposed by Rust, they do not have to worry about memory-related problems. Unlike the run time garbage collection, however, Rust does memory management at build time, through what is called the "borrow checker". A large part of this idea is "borrowed" (no pun intended ☺) from other language like C++, but Rust raises it to another level.

In Rust, as long as your program compiles, you do not have to worry about a certain class of memory-related errors. The program, *if you can build it,* is guaranteed to be free of this kind of errors.

Another unique feature of Rust is emphasis on expressions. Although Rust is an imperative language, most programming logic is carried out through expressions rather than statements. But, unlike expressions of (pure) functional programming languages, expressions in Rust can have side effects. Again, the separation of expressions vs statements is pretty unique in Rust. It lies somewhere between imperative languages and functional languages.

This book is a "mini language reference". It is a *reference* on the Rust programming language grammar, and it is not, for example, a tutorial on programming in Rust. It is a *mini* reference in that it does not cover all the language details in depth.

As suggested, if you have some experience with programming in any of the related languages, imperative or functional, you can pick up Rust rather easily by reading this reference. Alternatively, this book can also be useful to programmers who used before, or who are currently using, Rust. It can be used as a quick refresher of some of the essential concepts of Rust, or as an annotated cheatsheet, if you will. *This book is, however, not for real beginners.*

Dear Readers:

Please read this before you purchase, or start investing your time on, this book.

A programming language is like a set of standard lego blocks. There are small ones and there are big ones. Some blocks are straight and some are L-shaped. You use these lego blocks to build spaceships or submarines or amusement parks. Likewise, you build functioning programs by assembling these building blocks of a given programming language.

This book is a *language reference*, written in an informal style. It goes through each of these lego blocks, if you will. This book, however, does not teach you how to build a space shuttle or a sail boat. If this distinction is not clear to you, it's unlikely that you will benefit much from this book. This kind of language reference books that go through the syntax and semantics of the programming language broadly, but not necessarily in gory details, can be rather useful to programmers with a wide range of background and across different skill levels.

This book is not for complete beginners, however. When you start learning a foreign language, for instance, you do not start from the grammar. Likewise, this book will not be very useful to people who have little experience in real programming. On the other hand, if you have some experience programming in other languages, and if you want to quickly learn the essential elements of this particular language, then this book can suit your needs rather well.

Ultimately, only you can decide whether this book will be useful for you. But, as stated, this book is written for a wide audience, from beginner to intermediate. Even experienced programmers can benefit, e.g., by quickly going through books like this once in a while. We all tend to forget things, and a quick regular refresher is always a good idea. You will learn, or re-learn, something "new" every time.

Good luck!

Table of Contents

Chapter 1. Introduction

Rust is a general purpose programming language originally created for systems programming. Rust is widely used for Web applications and Web Assembly development, beyond system-level programming. Rust is a relatively new language, which is still rapidly evolving at this point. Rust currently has a *six week* release cycle.

Although we are primarily focusing on the Rust language in this book, we will briefly go through some non-language related aspects in this first chapter such as the basics of Rust software development. You can skip most of this chapter if you have some experience with programming in Rust.

1.1. Rust Tools

Rust provides pretty much all the standard tools when it comes to software development. For virtually all developers, the starting point is the *rustup* command. You can install it from the official download page:

- Install Rust - Rust Programming Language [https://www.rust-lang.org/tools/install]

1.1.1. *Rustup*

Rustup is used to manage Rust toolchains. A Rust *toolchain* is, roughly speaking, a combination of a Rust build from a particular channel (e.g., the Rust compiler and libraries) and a target platform(s). Rust has three standard channels, *nightly*, *beta*, and *stable*. Any feature, or commit, that is merged into the master branch is included in the daily *nightly* build. Every six weeks, a set of features are selected and a *beta* build is released. After six weeks, the *beta* build is then promoted to the *stable* build. Hence, there is a new *nightly* build every day, and there are new versions of *beta* and *stable Rust* every six weeks.

For example, on the author's computer,

```
$ rustup show
Default host: x86_64-unknown-linux-gnu
rustup home:  /home/harry/.rustup

installed toolchains                    ①
--------------------
stable-x86_64-unknown-linux-gnu (default)
beta-x86_64-unknown-linux-gnu

installed targets for active toolchain  ②
--------------------------------------
wasm32-unknown-unknown
x86_64-unknown-linux-gnu

active toolchain                        ③
----------------
stable-x86_64-unknown-linux-gnu (default)
rustc 1.69.0 (84c898d65 2023-04-16)
```

① On this particular computer, the *beta* and *stable* builds are installed, but not the *nightly* build.

② This computer has an *x86_64* cpu (Intel or AMD), and we are currently building Rust programs for the native platform only, other than the Web Assembly target (*wasm32*). Rust tools can be used for cross-architecture development.

③ Although we have both *beta* and *stable* toolchains, we are currently using the stable build by default. The current version of the Rust compiler (*rustc*) is *1.69*, as of this writing. Going through each tool, and their sub-commands and their options, is beyond the scope of this book. The readers who are new to Rust development are encouraged to consult the help/man pages of each tool. For example, you can start from *rustup -h*.

1.1.2. Rustc

Each Rust toolchain includes three essential tools, *rustc*, *cargo*, and *rustdoc*, among other things. *rustc* is the Rust compiler, which is no doubt the most important tool in Rust software development. But, in practice, *rustc* is rarely, if ever, used directly. Here's a simple usage example:

```
$ ls
main.rs
$ cat main.rs
fn main() {
    println!("Hello, world!");
}
$ rustc main.rs                          ①
$ ls
main  main.rs
$ ./main                                 ②
Hello, world!
```

① Again, try *rustc -h* for usage information and other available options.

② The *main* file is the build output (e.g., an executable) in this example.

1.1.3. Cargo

Cargo is a project and dependency management tool in Rust. Although it is an "optional" tool, it is the de-facto standard tool for building and managing Rust projects. Cargo uses *rustc* under the hood to compile the Rust source code.

If you are new to Rust development, *cargo init*, or *cargo new*, creates a new Cargo project. *cargo build* builds the project, e.g., for the target platforms of the currently active toolchain, and you can use *cargo run* to quickly build and run a program during development. A relatively new addition, *cargo add*, can be used to add

dependent packages to your project. (Cargo uses the Rust package repository, *crates.io*.) Try *cargo -h* for more information. For example,

```
$ pwd
/.../code/intro
$ ls                                    ①
$ cargo init                            ②
     Created binary (application) package
$ tree .                                ③
.
├── Cargo.toml
└── src
    └── main.rs
1 directory, 2 files
$ cat Cargo.toml                        ④
[package]
name = "intro"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]                          ⑤
$ cat src/main.rs                       ⑥
fn main() {
    println!("Hello, world!");
}
$ cargo build                           ⑦
   Compiling intro v0.1.0 (/.../code/intro)
    Finished dev [unoptimized + debuginfo] target(s) in 0.66s
$ ls                                    ⑧
Cargo.lock  Cargo.toml  src  target
$ cargo run                             ⑨
```

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/intro`
Hello, world!                                ⑩
```

① The current directory is empty.

② Try *cargo init -h* or *cargo new -h* for more information.

③ *cargo init* created two files, *Cargo.toml* and *src/main.rs*.

④ We briefly describe the Cargo project manifest file, *Cargo.toml*, below. Note that the project or package name, *intro* in this example, is taken from the name of the current directory. (See the *pwd* output above.)

⑤ This (new) project has currently no external crate dependency.

⑥ The default scaffolded file for the binary crate type.

⑦ Try *cargo build -h* for more information.

⑧ Note that the build output is stored in a directory named *target*. In general, this should not be included in the source code repository (e.g., by including this entry in the *.gitignore* file, if you are using git).

⑨ You can use *cargo run -q* if you want to display only the program output, without the compiler log messages. Try *cargo run -h*.

⑩ Yay, *Hello, world!*

1.1.4. Cargo manifest file

The Cargo package manifest file, *Cargo.toml*, uses the TOML file format, and it is similar to various project files used in other programming language ecosystems, e.g., *project.json* for a Node project and the *.csproj* file for a C# project, etc. As indicated in the comment in the scaffolded *Cargo.toml* file in the above example, you can find more information on the manifest file on the Rust reference website.

We discuss the `edition = "2021"` line in the next section on *Rust editions*.

1.1.5. Rustdoc

The *rustdoc* command generates a documentation for the given source code files. Try *rustdoc -h* for more information. But, as with *rustc*, the *cargo* command is usually used to indirectly invoke *rustdoc* for the current Cargo project.

For instance, using the same example,

```
$ cargo doc --open                          ①
 Documenting intro v0.1.0 (/.../code/intro)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
$ ls                                        ②
Cargo.lock  Cargo.toml  doc  src  target
```

① You can use *cargo doc -h* to get more information on this command. The *--open* flag opens the generated (HTML) documentation in the default Web browser after generating the docs.

② The documentation output is saved into the *doc* directory by default.

We briefly explain how to add *doc comments* to your source code in the next chapter, on lexical analysis.

1.2. Rust Editions

Rust has made some backward incompatible changes over the years since its first public release, version 1.0. In software engineering, backward incompatible changes are generally considered cardinal sins, especially for software that are as fundamental as coding languages.

But, sometimes, they are unavoidable. Rust's solution to this is to create new language variants every three years. These language variants are called the *editions* in Rust. Effectively, no changes in the Rust programming language

grammar are backward incompatible since those are *only* introduced to the new editions. The programs written in the older editions will never break. This strategy is not necessarily new or unique in Rust. JavaScript, for example, introduced a new (more modern) language variant through the use of the `"use strict"` declaration. Languages like C# include (effectively) a number of different language variants, e.g., using the configuration settings, etc.

Now, if you are starting a new Rust project, you should always use the most recent edition. The `2021` edition is the current edition, as of this writing, as illustrated by the example in the previous section. When Rust introduces a new edition, which is newer than the edition you are using for your project, you have two choices. Either you can keep it as is, or you can upgrade your project to use the newly defined edition. In general, you should try, or at least consider, upgrading your project unless there are any significant breaking changes that affect your project. This discussion is, however, beyond the scope of this book.

As shown earlier, a Rust package's edition is set in the *Cargo.toml* file using the `edition` entry. There are three editions at this point. *2015*, *2018*, and *2021*, and, for example, a new edition will be added in *2024*. We will not discuss the differences between different editions in this book. We only use the *2021* edition. (Incidentally, we do not discuss the "unsafe Rust" in this book either.)

1.3. The Language vs Standard Libraries

The separation between the core language and its standard libraries does not exist any more in the modern programming languages. Or, at least, the traditional dichotomy is no longer valid. The line between the language grammar and the supporting libraries has been increasingly becoming blurred (e.g., from the viewpoint of the application programmers).

This is especially true for Rust. The Rust programming language defines the syntax for a number of expressions and statements, and includes some primitive types, and so forth, but much of the language features come from the standard

libraries. Whether types like `Option` and `Result` are built into the language or not is (almost) completely irrelevant, to application programmers. They are truly part of the language. You cannot write any non-trivial programs beyond the "Hello World" without using these features.

The Rust language heavily relies, for example, on a number of *traits* defined in the standard library. In fact, they are so tightly integrated that certain language behavior is *defined* by those traits. Some of them are included in the Standard Library Prelude, and from the programmer's perspective, the distinction between *Builtin vs Prelude* has no real significance. Even beyond the Prelude, there are many traits and types, and functions, in various *modules* of the standard library that should be really viewed as an integral part of the language.

We include some of those core concepts in this book although *technically* they are not part of the language. On the flip side, we leave out so many of the important and widely-used types and traits in Rust, among other things. The readers are encouraged to consult appropriate references on the standard libraries.

1.4. Content

Rust is, despite its relatively short history, a rather complex language. This book covers all the essential, and fundamental, elements of Rust, including types, traits, variables, lifetimes, generics, functions, closures, patterns and pattern matching, statements, expressions, operators, operator overloading, basics of iterators, and error handling, as well as a few fundamental types and core traits, etc.

On the other hand, the book does not cover concurrency, async programming, many common collection types, type conversions, attributes, macros, IO functions, and runtime polymorphism in any depth, among many other things. We do not include unsafe Rust nor FFI (foreign function interfaces) either. Furthermore, we do not cover the "absolute basics" of programming in this book such as arithmetic, operator precedence, type inference, basic control flow using conditional and loop statements, etc. (In fact, control flow is done through expressions in Rust.)

Chapter 2. Lexical Elements

The programming language grammar generally has three layers, lexical structure, syntax, and semantics, from bottom up. We primarily focus on the Rust language syntax and semantics in this book. We briefly touch on the lexical structure of Rust programs in this chapter, however.

2.1. Input Format

As with most modern programming languages, Rust uses Unicode. In particular, the Rust lexer interprets a given Rust program as a sequence of Unicode characters encoded in *UTF-8*.

2.2. Comments

Rust supports both the C++-style line comments (`//...`) and the C-style block comments (`/* ... */`). In addition, Rust supports *doc comments* with a rich set of features. Comments are interpreted as a form of whitespace.

2.2.1. Line comments

A line comment starts from the character sequence `//` and it extends until the end of the line.

```
// I'm a line comment
const NAH: &str = "I'm not // a comment";      ①
const TRU: i32 = 42;  // I'm also a comment
///////// I'm also a line comment /////////
```

① The sequence `//` in a string literal does not start a comment.

2.2.2. Block comments

Unlike in most C-style languages, Rust supports "nested" block comments. (Why would anybody need nested block comments? *No comment.* ☺) For example,

```
/*                                  ①
I'm a comment.
/*
I'm a comment too.
*/                                  ②
I'm a comment three.
*/                                  ③
```

① It starts a block comment.

② This character sequence `*/` does not end the current block comment due to the intervening character sequence `/*` in the comment.

③ It ends the current block comment. Note that comments are not really *nested*. It's just that the character sequence `*/` can be included inside the Rust's block comment as long as it is paired with the matching `/*`.

2.2.3. Doc comments

Doc comments are used by the *rustdoc* command to generate code documentations. They are interpreted as a special syntax for builtin `doc` attribute. That is, Rust converts doc comments into semantically equivalent `doc` attributes. Attributes are briefly discussed later in the book. Some of the builtin attributes are also included, in the relevant chapters, throughout the book.

An outer line doc comment, `///...`, with exactly three slashes, or an outer block doc comment, `/** ... */`, precedes an item, and it provides the code documentation for the given item. ("Outer", in the sense that they are associated with their following items.)

```
/// I'm a mathematical pi.                ①
/// I'm very accurate.
pub const PI: f32 = 3.14;

/**                                       ②
 * I'm an edible pie.
 * I'm very edible.
 */
pub const PIE: &str = "American Pie";
```

① This is equivalent to a `doc` attribute, `#[doc="I'm a mathematical pi.\nI'm very accurate."]`. Note that multiple line comments form essentially a single doc comment.

② This is equivalent to `#[doc="I'm an edible pie.\nI'm very edible."]`. Note that the leading asterisk `*` and spaces in each line are ignored.

An inner line doc comment of the form `//!...` or an inner block doc comment of the form `/*! ... */` provides documentation for the parent item of the comment. They must appear before any other items in their parents. ("Inner", in the sense that they are associated with their enclosing items.)

```
1  //! I'm about this module.              ①
2  //! I'm about this module too.
3
4  pub fn demo() {
5      /*!                                 ②
6      I'm about the demo function.
7      */
8      /// I'm about X.                   ③
9      const X: i32 = 3;
10 }
```

① No other items in the module can precede this inner doc comment. This two-line doc comment is equivalent to the doc attribute, `#![doc="I'm about this module.\nI'm about this module too."]`.

② This inner doc comment provides the documentation for its parent, the `demo` function, and it is placed before any other items in the function body.

③ This outer line doc comment provides documentation for the following item, `X`, in the function body.

The Rust doc comment syntax allows the Markdown format, and it can be used to add markups, e.g., for emphasis and linking, etc., to the generated document. It also allows including code snippets through Markdown syntax, e.g., using the triple-backquote code blocks.

```
/// The `sum` function always returns 42.
/// ```                                    ①
/// # use comments::sum;                   ②
/// let a = sum(1, 2);                     ③
/// # print!("sum = {a}");
/// assert_eq!(a, 42);                     ④
/// ```
pub fn sum(_a: i32, _b: i32) -> i32 { 42 }
```

① A code block. This starting ```` ``` ```` is equivalent to ```` ```rust ````.

② The lines starting with a hash symbol, `#`, are not included in the generated doc.

③ The code example in the doc comment can be "executed", e.g., for testing, or for verifying the validity of the code. The lines with the leading hashes are also executed in this context.

④ You can run the example code in the doc comment in a couple of different ways. For example, *cargo test --doc* will test this code snippet. (Note that you can only run unit tests on library crates.)

2.3. Tokens

The lexer reads the Rust source input and produces a sequence of *tokens*, e.g., after removing whitespaces and what not. The parser then reads these lexical tokens and creates an abstract syntax tree (AST) for further processing.

Tokens in Rust belong to one of the following six categories:

- Delimiters,
- Operators and other punctuation symbols,
- Identifiers,
- Keywords,
- Lifetimes, and
- Literals.

2.4. Delimiters

Rust uses three kinds of brackets as delimiters. An open bracket must always be paired with the corresponding close bracket.

- Parentheses, `( )`,
- Curly braces, `{}`, and
- Square brackets, `[]`.

They are used in many different contexts. For example, parentheses are used for expression grouping and function calls, and curly braces are used for expression blocks, among other things. Their specific uses are explained in the relevant chapters throughout the book.

2.5. Operaotrs and Other Punctuation

The following punctuation symbols are used for operators:

```
+   -   *   /
%   ^   !   &   |
&&  ||  <<  >>
+=  -=  *=  /=  %=
^=  &=  |=  <<=  >>=
=   ==  !=  >   <   >=  <=
```

Other punctuation symbol tokens used in Rust:

```
@   _    .    ..   ..=  ...
,   ;    :    ::
->  =>   #    $    ?
```

Uses of some of these tokens, including operators, are explained in the relevant part of this book. Otherwise, they generally have the same usual meanings as in most other C-style programming languages. (E.g., the `+` symbol is usually used as an arithmetic addition operator, possibly in addition to other uses. In Rust, the `+` token is also used in the trait bounds, among other things.)

2.6. Identifiers

In general, Unicode *letters* (e.g., Unicode Standard Annex #31), numbers `0` through `9`, and underscore character `_` are allowed in identifiers. (There are some exceptions, but we will not go through them in this book.) An identifier cannot start with a number.

Identifiers may not be a strict or reserved keyword, but they can be "escaped" using the raw identifier syntax (except for `crate`, `super`, `self`, and `Self`). A raw identifier is prefixed by `r#` (which is not part of the identifier). For example,

```
fn demo() {
  let r#let = 333;                    ①
  let doubled = r#let * 2;
  println!("{let} * 2 = {doubled}");   ②
}
```

① `let` is a strict keyword, and we use the raw identifier syntax here.

② Note that, in this context, `let` refers to the variable declared in the current scope, and not the keyword `let`. This will print out *333 * 2 = 666*.

The single underscore name `_` has special uses in Rust. Otherwise, identifiers starting with an underscore are only used, by convention, in places where they are intentionally unused (e.g., temporarily during the development). The Rust compiler will not issue the unused warnings for those identifiers. For example,

```
fn repeat(s: &str, times: i32) -> String {
  todo!();
}
```

The `todo!` macro takes care of returning a placeholder `String` value, etc. But, the compiler, with `#[warn(unused_variables)]`, will still issue warnings for unused variables. However, there will be no warnings for the following version.

```
fn repeat(_s: &str, _times: i32) -> String {
  todo!();
}
```

2.7. Keywords

Rust keywords belong to one of the following three categories:

Strict Keyword

Strict keywords can only be used in particular syntactic contexts where they are specifically designed for. In particular, these keywords cannot be used as the names of any items, variables and function parameters, fields and variants, type parameters, lifetime parameters or loop labels, macros or attributes, macro placeholders, and crates.

```
as       async     await     break
const    continue  crate     dyn
else     enum      extern    false
fn       for       if        impl
in       let       loop      match
mod      move      mut       pub
ref      return    self      Self
static   struct    super     trait
true     type      unsafe    use
where    while
```

As to what these keywords mean, we will explain them as they appear in the relevant contexts throughout the book.

Weak Keywords

These keywords have special meaning only in certain contexts, and they can be, for example, used as variable or function names.

```
'static  union     macro_rules
```

Reserved Keywords

These identifiers are not currently used as keywords, but they are reserved for possible future use. Reserved keywords have the same restrictions as strict keywords.

```
abstract become    box       do
final    macro     override  priv
try      typeof    unsized   virtual
yield
```

2.8. Lifetimes and Loop Labels

A lifetime token comprises a single quote/apostrophe character (`'`) followed by any valid identifier or keyword. `'_` is also a valid lifetime token. The lexer accepts lifetime tokens as valid tokens, and hence they can be used, for example, in macros. Lifetime tokens that comprise non-keyword identifiers after the `'` prefix can be used as lifetime parameters and loop labels. For example,

```
fn fst<'a>(x: &'a str, _y: &'a str) -> &'a str {
  x
}
```

Loop labels are used with `break` and `continue` expressions.

```
fn label_demo() {
  'label: for i in 0..10 {
    if i > 5 { break 'label }
  }
}
```

2.9. Literals

Literals are special kinds of tokens that represent constant expressions of certain builtin types. Furthermore, literals postfixed with (non-raw) identifier suffixes are also considered lexically valid *literal tokens* in Rust (although they may or may not correspond to syntactically valid literals, e.g., representing constant expressions). The literal tokens with suffixes that are not valid literal expressions in Rust are typically used for macro processing. The macro implementations can decide what to do with those tokens.

Rust itself only considers the following as valid literals. In the interest of space, we only briefly describe them here. Other references can be consulted for more detailed lexical and syntactic rules. If you are familiar with any C-style languages, they all have rather similar literal syntax, including Rust.

2.9.1. Boolean literals

The literal tokens, `true` and `false`, represent the two values of the `bool` type, the logical true and false, respectively.

2.9.2. Number literals

A number literal can be either integer literal or floating-point literal. Rust includes 12 primitive types, `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, `u128`, `i128`, `usize`, and `isize` for integers, and 2 types, `f32` and `f64`, for floating-point numbers.

2.9.3. Integer literals

Rust supports the four integer literal forms that are commonly found in most other programming languages, namely, decimal literals, hexadecimal literals, octal literals, and binary literals, with essentially the same lexical rules. The type of an integer literal can be inferred through Rust's type inference rules. Alternatively,

the type of a literal can be explicitly set using one of the integer literal suffixes, `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, `u128`, `i128`, `usize`, or `isize`, corresponding to one of the integral types. For example,

```
use std::any::{Any, TypeId};
fn integers() {
  let a: u64 = 0;                          ①
  assert_eq!(TypeId::of::<u64>(), a.type_id());
  let b = 0x42;                            ②
  assert_eq!(TypeId::of::<i32>(), b.type_id());
  let c = 0o7000i16;                       ③
  assert_eq!(TypeId::of::<i16>(), c.type_id());
  let d = 0b1010_0000_u8;                  ④
  assert_eq!(TypeId::of::<u8>(), d.type_id());
}
```

① The integer literal `0` is assigned to a `u64` variable.

② The type of an integer literal is `i32` by default. `0x42` is a hex literal.

③ `0o7000i16` is an octal literal with the `i16` suffix. Hence, its type is `i16`. Note that if the number is too big to fit into the specified type, it is a static type error.

④ `0b1010_0000_u8` is a binary literal of the `u8` type. All number literals, both integer and floating point literals, allow `_` as a visual separator. They do not affect the value of the literal.

2.9.4. Tuple index

A certain subset of integer literals can be used as a tuple index, which refers to a field of tuples, tuple structs, or tuple variants. In particular, decimal literals without the literal suffix that is equal to, or bigger than, `0` are lexically valid tuple indices. For example,

```
fn tuple_indices() {
  let crabby = ("crab", "lobster");      ①
  let crab = crabby.0;                   ②
  let lobster = crabby.1;                ③
  println!("Crab is {crab}.");           ④
  println!("Lobster is {lobster}.");
}
```

① Tuple types are described later.

② Syntactically, tuple indices start with `0` and they increment by `1`.

③ For example, `crabby.1` refer to the second element of the tuple, `crabby`.

④ This will output *Crab is crab.*

2.9.5. Floating-point literals

Rust's floating-point literals are again more or less the same as those found in other similar languages, except that, in Rust, the type-specifying literal suffix, either `f32` or `f64`, can be used with a floating point literal. For example,

```
use std::any::{Any, TypeId};
fn floats() {
  let a: f32 = 10.;                      ①
  let b = 100.;                          ②
  println!("a + b = {}", a + b);
  assert_eq!(TypeId::of::<f32>(), b.type_id());
  let c = 3.14159265358979;              ③
  assert_eq!(TypeId::of::<f64>(), c.type_id());
  let d = 3E+15f32;                      ④
  assert_eq!(TypeId::of::<f32>(), d.type_id());
  let e = 1_0__0___0____f32;             ⑤
  assert_eq!(TypeId::of::<f32>(), e.type_id());
```

```
}
```

① A floating-number literal without a literal suffix needs to include a period (`.`) or the exponent symbol `e` or `E`. The float literal `10.0` is assigned to the variable `a` of the `f32` type in this example.

② The type of `b` is inferred to be `f32` because of the addition in the next line. In Rust, only the values of the same type can be added, and hence `b` has to be `f32`.

③ In the under-constrained context, the type of a float literal is `f64` by default.

④ A floating-point literal with an exponent. It has the literal suffix `f32` and hence its type is `f32`.

⑤ Like the integer literals, float literals can include `_` as visual separators. The type of `e` is `f32`, and its value is `1000.0`. Note that this literal does not require a period or `e`/`E` since its type is explicitly specified using the literal suffix.

2.9.6. Character literals

A character literal is a *single* Unicode character, e.g., encoded in UTF-8 in the source, which is enclosed in a pair of single quotes (`'`). The single quote itself needs to be escaped as `'`. Rust also supports C-style escape sequence syntax for a few well-defined set of characters such as the newline, e.g., `'\n'`, and tab characters, e.g., `'\t'`.

2.9.7. String literals

A string literal is a sequence of characters enclosed in a pair of double quotes (`"`). The double quote character needs to be escaped as `\"`. String literals can include escape characters.

In Rust, line breaks, e.g., `\n` and `\r\n`, can be included in a string literal without escaping, and they become part of the string value. To escape a line break, one can put a backslash (`\`), again without escaping, just before the line break. In such a

case, all leading white spaces in the next line, if any, are also ignored along with the backslash and line break characters. For example,

```
fn strings() {
  let emily = "I heard a fly buzz -
    When I died.";                          ①
  println!("{emily}");
  let william1 = "My heart leaps up";       ②
  let william2 = "My heart \
                   leaps up";               ③
  println!("{william1} (vs) {william2}");
}
```

① Rust's string literals are naturally "multiline strings". Note the four leading spaces in the second line. Those are part of this string literal.

② This string literal ...

③ ... and this string literal are equivalent. Note that, although you cannot see (and hence some people prefer not to use this kind of syntax), there is no trailing white spaces, other than the newline, after the backslash character.

2.9.8. Raw string literals

Unlike the (normal) string literals, raw string literals do not process escape characters. They start with a character sequence, the prefix `r`, any number of hashes `#`, and a double quote `"`, and it ends with a character sequence comprising a double quote and *the same number of hashes*.

Raw string literals can include special characters such as `\` without escaping. For example, `"` and `\\` are considered both two-character sequences, `\` followed by `"` and `\` followed by another `\`, respectively. The only limitation is that a raw string literal cannot include its closing character sequence, for the obvious reason.

For example,

```
fn raw_strings() {
  let raw1 = r"Rust is your oyster.";     ①
  let raw2 = r#"I love "oyster"."#;       ②
  let raw3 = r##"Oyster is #1 \
  the most loved food by the crabs."##;   ③
  println!("{raw1}\n{raw2}\n{raw3}");
}
```

① This raw string literal starts with `r"` and ends with `"`.

② This literal starts with `r#"` and ends with `"#`.

③ This literal starts with `r##"` and it ends with `"##`, and hence it cannot include the same character sequence `"##`. Note that the backslash and newline on the first line and the two leading spaces on the second line are part of the string.

2.9.9. Byte literals

A character in the ASCII range can be used to represent the corresponding byte. This is called the *byte literal*, and it is lexically similar to the character literals, except that it uses the prefix `b`. The single quote character needs to be escaped as `b'\''`. For example, `b'a'` is a byte literal corresponding to `97`. (The ASCII code of `'a'` is a decimal number `97`.) As a matter of fact, a byte in Rust is equivalent to an unsigned 8-bit integer number (`u8`).

```
fn byte_literal() {
  let b1 = b'A';                          ①
  let b2 = 65;                            ②
  assert_eq!(b1, b2);                     ③
}
```

① The type of `b1` is `u8`, the unsigned 8-bit integer type.

② The type of `b2` is inferred to be `u8`, because of the statement in the next line. Rust's type inference uses contextual information, and, for example, not just the declaration statement.

③ Both `b1` and `b2` have the same type, `u8`, and they have the same value, `65u8`. And, hence this assertion will succeed.

2.9.10. Byte string literals

Just like a string literal represents a sequence of characters, a *byte string literal* represents a sequence of bytes. The byte string literal is lexically similar to string literals, but it starts with the prefix `b`, and it can only contain bytes, or ASCII characters. The type of a byte string literal of length `n` is `&'static [u8; n]`.

For example,

```
fn byte_strings() {
    let s1 = b"Hello' \"World\"!";       ①
    let s2 = &[                          ②
        b'H', b'e', b'l', b'l', b'o', b'\'', b' ',
        b'"', b'W', b'o', b'r', b'l', b'd', b'"', b'!',
    ];
    assert_eq!(*s1, *s2);                ③
}
```

① A byte string literal, consisting of 15 bytes.

② A shared reference (`&`) to an array, consisting of 15 *byte elements*. Note the escape byte syntax.

③ This assertion will succeed since these two values have exactly the same type and the same value. In fact, `assert_eq!(s1, s2)` will also succeed.

2.9.11. Raw byte string literals

A *raw byte string literal* is again lexically similar to a raw string literal, except that

- It starts with the prefix `b`, followed by the raw string prefix `r`, and
- It can only include ASCII-range characters, which represent bytes (or, `u8`).

Hence, raw byte string literals start with the prefix `br`, any number of hashes `#`, and a double quote `"`, and they end with a double quote and the same number of hashes. The type of a raw byte string literal of length `n` is also `&'static [u8; n]`. For example,

```
fn raw_byte_strings() {
  let crab1 = br"You cannot teach a crab
  to walk straight";
  let crab2 = br##"~`!@#$%^&*(){}[]<>\|\ ①
;:'"#,.\n\t\'\"\\?/\"##;                  ②
  let len1 = crab1.len();
  let len2 = crab2.len();
  println!("Crab1 ({len1}): {crab1:?}"); ③
  println!("Crab2 ({len2}): {crab2:?}"); ④
}
```

① This literal starts with `br##"` and ends with `"##`.

② Note that `\n`, for example, is counted as two bytes in the raw byte string literals, that is, `\` followed by `n`. It is not an escape sequence, e.g., a newline.

③ This `println!` macro statement will print, *Crab1 (42): [89, 111, ..., 116]*. For reference, `"You cannot teach a crab to walk straight".len()` is `40`.

④ Likewise, this will print, *Crab2 (42): [126, 96, ..., 92]*. In both `crab1` and `crab2`, the (invisible) newline characters at the end of their first lines are part of the raw byte string literals.

Chapter 3. Using Attributes

Rust uses *attributes* to affect various core aspects of the language as well as the build processes, including compilation and diagnostics, etc. This is a common pattern used in many different programming languages, and most people reading this book should be familiar with them. For example, Rust's attribute system is originally based on that of C#. Other languages like Java, Python, and TypeScript also support similar constructs. Rust's attributes can be classified into the following four categories:

- Builtin attributes,
- Macro attributes,
- Derive macro attributes, and
- Tool attributes.

We will go over some of the builtin attributes of Rust in the special sections named "Builtin Attributes" throughout this book. You can skip them if you are only interested in reviewing the language grammar. In general, an attribute adds a certain type of free-form metadata to the program, or more specifically, to a particular item or crate targeted by the attribute, and it affects the generation of the build output in some way. Syntactically, attributes are enclosed in square brackets (`[]`), and there are two kinds of attributes:

Inner attributes

An inner attribute is written with a hash followed by a bang (`#!`). They apply to the item that *encloses* the attribute. Note that an attribute applied to a whole crate can only use the inner attribute syntax, e.g., `#![crate_attribute]`.

Outer attributes

An outer attribute is starts with a hash (`#`), without the bang. They apply to the thing that *follows* the attribute. For example, `#[item_attribute] ITEM`.

Here are some examples:

```
#![crate_type = "lib"]                    ①
```

① The attribute `crate_type` is used to set the create type, *bin* vs *lib*. This inner attribute applies to the *entire crate*.

```
#![allow(unused_variables)]               ①
```

① A module-level attribute, to suppress the `unused_variables` *lint warning*.

```
fn some_function() {
  #![allow(non_camel_case_types)]         ①
  let _crab_case = ();
  let _lobster_case = ();
}
```

① Another *lint attribute* applied to the whole function.

```
#[test]                                   ①
fn test_cancer() { /* ... */ }
```

① It indicates that the attributed function is a unit test.

```
#[cfg(target_os = "linux")]               ①
mod driver { /* ... */ }
```

① A conditionally-compiled module. We do not cover the `cfg` attribute in this book, but it will be worthwhile to look it up if you are just starting with Rust programming.

Chapter 4. Using Macros

Macros provide a way to *extend* the functionality and syntax of Rust. They are defined and invoked through consistent and type-safe syntax. There are two ways to define new macros:

- Declarative macros: Simple macros can be created in a declarative way, e.g., using the `macro_rules!` macro.
- Procedural Macros: One can define a function that takes an input token stream and outputs a transformed token stream to create function-like macros, custom derives, and custom attributes.

When a macro is invoked, with a sequence of input tokens, it is expanded and ultimately replaced with its result, e.g., in the abstract syntax tree. This happens at compile time. Macros may be invoked in the following situations:

- Patterns,
- Types,
- Expressions and statements,
- Items and associated items, and
- `macro_rules!` transcribers.

Macros are rather commonly used in Rust programming, for various reasons. For example, Rust does not support vararg functions, and we can define and use a function-like macro that takes an arbitrary number of "arguments". Macros are often used to reduce boilerplate code. They are also used to affect various aspects of Rust programs and even the build process. And so on and on.

We will not discuss macros any further in this book, e.g., as to how to create your own macros, etc. Instead, we will just list a few commonly used macros in this chapter for the readers who have had a limited exposure to Rust.

4.1. The `format!` Macro

The `format!` macro creates a `String` based on a provided *format string*, e.g., in the form of a string literal, and other runtime expressions. It essentially does a string interpolation. The format string uses the `{}` token as placeholders. There are three different forms.

```
fn format_macro_demo() {
  let s1 = format!("{} {} {}", 1, 2, 3); ①
  println!("{}", s1);
  let s2 = format!("{0} {0}", 10);       ②
  println!("{0}", s2);
  let s3 = format!("{a}", a = 666);      ③
  println!("{s3}");
}
```

① Each of the `{}` tokens in a format string is matched to each of the parameters following the format string, e.g., one by one, in sequence.

② The indices included in the `{index}` refer to the following parameters, e.g., from left to right.

③ Named arguments can also be used. For instance, in this example, `a` inside `{a}` refers to the variable defined in the parameter list. The order is not important.

4.2. The `print!` and `println!` Macros

The `print` macros use the same convention as the `format!` macro. They take a format string as a first argument and create an interpolated string using the trailing arguments. Instead of returning the result string, `print!` and `println!` prints out the result to the standard out. The `println!` macro additionally appends a newline at the end.

4.3. The `eprint!` and `eprintln!` Macros

The `eprint` macros work the same way as `print` macros. But, instead of outputting the result to the standard out, they print the output to the standard err. Similar to `println!`, the `eprintln!` macro adds a trailing newline.

4.4. The `write!` and `writeln!` Macros

The `write!` and `writeln!` macros are likewise similar to `print!` and `println!` macros, respectively. But, they take an output stream as their first argument, and emit the formatted string to that stream.

A simple illustration:

```
use std::io::Write;
fn write_macro_demo() -> Result<(), Error> {
  let mut buf = Vec::new();
  write!(&mut buf, "Hello {} times!", 42)?;
  // ...                                    ①
  let text = String::from_utf8_lossy(&buf);
  println!("{text}");
  Ok(())
}
```

① Now that we have a `Vec<u8>`, we can do something with it.

4.5. The `panic!` Macro

The `panic!` macro unconditionally causes a runtime panic, which immediately terminates the current thread. If the main thread panics, then it will terminate all other threads and end the program.

One can provide a string argument in the `panic!` macro, which is provided to the caller. For example,

```
fn panic_macro_demo_1() {
  panic!();                                ①
}
fn panic_macro_demo_2() {
  panic!("Long time no see!");             ②
}
fn panic_macro_demo_3() {
  panic!(
    "Well, it's been {} years.", 1000      ③
  );
}
```

① The trailing pair of parentheses is required when invoking function-like macros even if there are no arguments to pass.

② The `panic!` macro can take a string argument.

③ In fact, it can also use string formatting using the `format` macro-like syntax.

If we run the following main program, for example, it will panic.

main.rs

```
fn main() {
  panic_macro_demo_2();
}
```

```
$ cargo run -q
thread 'main' panicked at 'Long time no see!', src/main.rs:6:3
```

4.6. The `assert!`, `assert_eq!`, and `assert_ne!` Macros

The `assert` macros are used to invoke the `panic!` macro when certain conditions are, or they are not, met at run time.

The `assert!` macro

The `assert!` macro takes a boolean expression, and it asserts that its value is `true`. That is, if the expression evaluates to `false` at run time, it will panic.

The `assert_eq!` macro

This macro takes two expressions, and it asserts that they are equal to each other (using `PartialEq`). On panic, it will print the values of the expressions.

The `assert_ne!` macro

It asserts that the given two expressions are not equal to each other.

All three assert macros take an optional string expression as the last argument, which will be used as a custom panic message. They can also be used with format strings, e.g., with any trailing arguments. For example,

```
fn assert_macros() {
  assert!(true == true, "True must be true");
  let t = 333;
  assert_eq!(333, t, "How can 333 be not {t}?");
  assert_ne!("hello", "world", "Is hello world?");
}
```

4.7. The dbg! Macro

The dbg! macro evaluates a given expression, it prints the value to standard error, and it returns the value. This macro is mainly used for quick and dirty debugging during development. For example,

```
fn dbg_macro_demo() {
  let a = 963;
  let x = dbg!(a + 36);                    ①
  assert_eq!(x, 999);
  let b = String::from("Rust");
  let y = dbg!(&b);                        ②
  println!("{y}");
}
```

① This dbg! macro invocation will print something like this to standard error (stderr), *[src/main.rs:6] a + 36 = 999*.

② The dbg! macro can take ownership of the given expression, and hence it is often better to invoke it with a borrowed reference (&), e.g., not to affect the normal flow of the program.

4.8. The todo! Macro

The todo! macro is commonly used as a placeholder. It indicates unfinished code. This macro is especially useful in the early stage of development when you just need to make your code compile, etc. Note that it will *always* panic at run time.

```
fn todo_macro_demo(x: i32, y: f32) -> Option<bool> {
  todo!("to do or not to do?");            ①
}
```

① Note that, although the function is supposed to return a value of `Option<bool>`, this code still compiles. The `todo!` macro does the magic.

4.9. The `unimplemented!` Macro

The `unimplemented!` macro is generally used to indicate a code that is not implemented, e.g., because you are just prototyping, or because you don't plan to provide actual implementations for things that are otherwise required syntactically. For example, they are commonly used when you are implementing only a subset of the required methods of a trait. This macro, when invoked, panics with a message, *not implemented*.

The difference between `unimplemented!` and `todo!` is that they indicate different intentions. The `todo!` macro clearly indicates that the code is to be implemented, whereas `unimplemented!` does not make such a claim. Furthermore, IDEs may use `todo` entries for various purposes.

4.10. The `vec!` Macro

This is also one of the most commonly used macros. To create and initialize an instance of the `Vec` type, we will generally have to go through multiple steps. The `vec!` macro allows a `Vec` to be defined more easily. For example,

```
fn vec_macro_demo() {
  let _v = vec![1, 2, 3];                ①
  let _v = vec![10; 3];                  ②
}
```

① It creates a `Vec` with the specified elements.

② It creates a `Vec` from a given element (e.g., `10`) and size (e.g., `3`).

Chapter 5. Rust Programs

A *Rust program* is generally written in one or more source files, and they are compiled into either a *binary crate* or a *library crate*. We will go through the high-level structures of Rust programs in this chapter, both physical and logical.

5.1. Source Files

Rust source code files have the extension `.rs`. A Rust source code is lexically a sequence of character-based tokens, as we briefly discuss in the previous chapter, Lexical Analysis. Syntactically, a Rust source file comprises a sequence of *item* definitions such as *modules* and what not. It may include optional *inner attributes* in the beginning.

A source file that is the starting point of compilation, e.g., through *rustc myprog.rs*, corresponds to a *crate module* (of that build output), which can contain other dependent *module* definitions, and other *items*, either in the same or other source files. In general, the module of a source file, and its name and its location in the *module tree*, may be defined by its referencing (or, "parent") source file.

5.2. Crates

Crates are the basic units of Rust programs. A crate, as a whole, is used for building, distributing, and runtime loading. The Rust compiler takes, as an input, a single Rust source file of a source crate (and, indirectly other files in the crate), and it produces an executable or a library crate as an output. A crate contains one or more modules, organized in a tree-like hierarchy. The top level module is anonymous, and all other modules have canonical module paths within the tree. (If you are new to Rust programming, it can take a little while to get used to this dual structure of the crate's physical (files) and logical (modules) hierarchies. We will go through the modules in some more detail later in this chapter.)

Builtin Attributes

The `crate_name` Attribute

The Rust compiler *rustc*, by default, uses the (top-level) source file name as the crate name. But, it can be overridden by the command line flag *--crate -name*. Or, if you use *cargo*, you can set the crate name using the *name* field in the *[package]* section. Note, however, that these options are not part of the language specification per se. We mostly focus on the Rust language in this book (although there are some exceptions).

As indicated, Rust's compilation process starts by taking a *single* target source file, which corresponds to a crate, and which may end up loading other source files as dependent modules. We can include the `crate_name` attribute, at the top of this top-level source file (e.g., as an *inner attribute*), to set the target crate's name. For example,

my-fantastic-rust-program.rs

```
#![crate_name = "my_fantastic_crate"]    ①

// Other items in the source file here...
```

① The dashes, `-`, seem to be more commonly used in file names (on certain platforms) and package names on *crates.io*, , e.g., as word separators. But, dashes are not valid characters in identifiers (e.g., the crate or module names). If you specify a dash-separated *name* in Cargo.toml, then Cargo seems to automatically, and implicitly, "normalize" it, if you will, to the corresponding name with underscores. Hence, in this particular context, dashes and underscores are mostly interchangeable. For example, if we publish this crate with a package name, *my-fantastic-crate*, to crates.io, other developers can use it with the crate name

> *my_fantastic_crate*. This is cargo-specific, and as far as we know, it is not a part of the language specification. Some Rust developers simply do away with dashes, e.g., to avoid any potential complications.
>
> As stated, although this kind of information can be rather important in practice when you develop Rust programs, it is beyond the scope of this reference, which covers the core language features, and, in fact, only a subset. The readers are encouraged to consult other resources for more practical information that is not covered in this book.

5.3. Items

A source crate contains what is called the *items* in Rust, which are processed at compile time. All items belong to modules, which themselves are also items. As stated, modules in a crate are organized in a tree-like structure, starting from the single top-level anonymous module corresponding to the crate. All other items within the crate have paths within the module tree.

The following are items:

- Modules,
- `use` declarations,
- Function definitions,
- Trait definitions,
- Type aliases,
- The `struct` definitions,
- The `union` definitions,
- The `enum` definitions,

- Implementations (`impl`'s),
- Constant items,
- Static items,
- `extern` crate declarations, and
- `extern` blocks.

We will briefly go through some of these items, in this and the next few chapters, including the Items chapter. Some of the more important items, along with other essential language constructs like statements and expressions, are explained in more detail, and referenced, throughout this book.

5.4. Modules

A module is a container for items, including other modules. Modules can nest. As indicated, a crate defines, or corresponds to, an implicit anonymous module, which may be called the crate module, or top-level module, etc. All items in a crate implicitly belong to this crate module. For example,

hello.rs

```
pub const A: &str = "A";                    ①
```

① The item `A` belongs to the crate module.

Otherwise, new named module items can be introduced into the tree of modules of the given crate using module declarations. More specifically, a module can be defined using the `mod` keyword in the following two different ways:

- A module with a body, e.g., in the same referencing file, or
- A module with its body in a separate file.

5.4.1. A module with a body

In the first syntactic form,

```
mod MOD-NAME {
  ITEMS
}
```

A module is declared with a name, denoted by `MOD-NAME` in this notation, followed by a pair of curly braces, which forms a body block. Inside this block, zero or more other items, denoted by `ITEMS` here, can be included.

Note that all-caps names are placeholders in this (rather informal) notation, as they are used throughout this book. That is, proper syntactic elements, e.g., identifiers or statements, etc., need to be substituted for these placeholders to create syntactically valid Rust code. The readers are encouraged to look up more precise grammar, e.g., written in EBNF, if necessary.

For example, in a file named *hello.rs*,

hello.rs

```
mod my_mod {
  pub const A: &str = "I'm an A";
}
```

This declaration introduces a new module, with a name `my_mod`, *as a child* of the current module (that corresponds to the *hello.rs* source file, which may or may not be the top-level file for a crate). This module includes one item, e.g., a public constant, in this example.

The declaring, or parent, module can refer to the items in this inline module using the (relative) path syntax, as we describe in the next chapter. Using the same example module, `my_mod`, in the *hello.rs* file, for instance,

hello.rs

```
mod my_mod {
  pub const A: &str = "I'm an A";
}

fn print_a() {
  println!("{}", my_mod::A);              ①
}
```

① When the `print_a` function is called, this statement will print *I'm an A*. The keyword `fn` is used to declare a new function item in Rust. The commonly used macro function `println`, and how it works, is explained a bit later in the book, if you are totally new to Rust.

Note that modules and types share the same namespace in Rust, but it becomes rarely an issue. By convention, we use the snake case names for modules and functions and the upper camel case (or, pascal case) for types and variants, which help avoid name conflicts.

5.4.2. A module in a file

In the alternative syntax, a named module can be declared without a body.

```
mod MOD-NAME;
```

A module without a body, or its content, is loaded from an external file. Where exactly that file is located depends on various factors.

First, the `path` attribute can be used for loading nested external file modules, e.g., by specifying the file path. The file path is generally relative to the directory where the declaring source file is located. But, more commonly, in the vast majority of cases, we simply use a set of conventions, integrated into Cargo. We will use the *newer* convention, which is the recommended way moving forward.

At the top-level, as stated, the name of the crate module is essentially the same as the crate name. When a module declares another (child) module, the child module name is used as the base of the external module file. For example,

world.rs

```
mod my_mod_b;
```

The (parent) module in the *world.rs* file declares a module named `my_mod_b` in this example. Then, by default, the body of this module is found in a file named *my_mod_b.rs* in the same directory as *world.rs*. For instance,

my_mod_b.rs

```
pub const B: &str = "I'm a Bee";
```

Then, the items in this module can be referred to in the same way as before, e.g., using the path syntax. In fact, the two module declaration forms, with and without a body, are semantically equivalent. Here's an updated *world.rs* file,

world.rs

```
mod my_mod_b;                          ①
fn print_b() {
  println!("{}", my_mod_b::B);          ②
}
```

① The nested module declaration, without a body, as before.

② An example usage. When `print_b` is called, this will print *I'm a Bee*.

One thing to note is that, due to the way how the older convention works (which we do not describe in this book), one cannot have two files named `./abc.rs` and `./abc/mod.rs` at the same time. For instance, we cannot have *./my_mod_b/mod.rs* in this particular example since we already have *./my_mod_b.rs*.

If you are new to Rust, and especially if you are coming from other programming languages, this can be somewhat confusing. The Rust module system works rather differently than in most other languages. Note, for instance, that the *my_mod_b.rs* file does *not* declare the *my_mod_b* module. It is declared in its parent module, e.g., *world.rs* in this example. The *my_mod_b.rs* file includes the body of the *my_mod_b* module (which can, incidentally, include its own child module declarations, etc.).

Builtin Attributes

The `path` Attribute

The `path` attribute affects the directories and files used for loading external file modules. In the *newer* convention, the file path is always relative to the directory which the source file is in. For example,

src/a/seafood.rs

```
#[path = "delicious-crab.rs"]          ①
mod crab;                              ②
```

① The `path` attribute is, or is used as, an outer attribute, which affects the item following the attribute, e.g., the `crab` module in this example.

② The body of the nested module `crab` is included in the *src/a/delicious-crab.rs* file. The *path* of this module is `crate::a::seafood::crab`.

As another example,

src/a.rs

```
mod seafood {                                ①
  #[path = "delicious-crab.rs"]              ②
  mod crab;                                  ③
}
```

① A module declaration with its body in the declaring file. The module corresponding to the *a.rs* file may be referred to as `crate::a`. Then this `seafood` module has a full path, `crate::a::seafood`.

② This attribute indicates that the body of the nested external file module `crab` is included in the *src/a/seafood/delicious-crab.rs* file.

③ The full path of the `crab` module is `crate::a::seafood::crab`. The path is further discussed in the next chapter.

5.5. The `main` Function

A Rust binary crate should (normally) contain a `main` function, which is the entry point to the program. `main` should have one of the following two signatures:

```
fn main() -> ();                             ①
```

① `()` refers to the unit type. That is, it indicates that the function returns (`->`) nothing. This signature is equivalent to `fn main()`.

The `main` function can also have the following signature:

```
fn main() -> Result<(), E: Error>;        ①
```

① The `Result` type is described later in the book.

Alternatively, one can return the program exit code directly to the caller (e.g., the operating system) by calling the `std::process::exit(code: i32)` function, etc. Some examples are given at the end of the book. But, it should be noted, in general, that Rust is a constantly evolving language, and the content provided in this book may not be completely up-to-date or even entirely accurate. For example, we describe one additional way of declaring `main`, which was stabilized in Rust 1.62, at the end of the book, in the error handling chapter.

Builtin Attributes

The `no_main` Attribute

In a special circumstance, e.g., when `main` is defined in some other object being linked to, one can use the `no_main` attribute to disable emitting the `main` symbol for an executable binary output. This attribute can be applied at the crate level. For example,

my-main-program.rs

```
#![no_main]
// The rest of the code without the main function ...
```

Chapter 6. Names and Paths

A Rust program comprises various *entities* that can be referred to in the source code, e.g., using their *names*, or *paths*. Entities include types, items, lifetimes, variables, fields, and attributes, etc. Various declaration statements introduce new names for entities. Entity names are valid within a lexically defined scope. Some names are implicitly declared in the language, and they can be referred to throughout the lifetime of the program. In Rust, names are segregated into a few different namespaces, based on their entity kinds, and the same names belonging to different namespaces do not cause name conflict.

6.1. Paths

Paths are used to refer to certain items or local variables. Syntactically, a path is a sequence of one or more optional *path qualifiers* followed by one or more (non-qualifier) path segments, separated by namespace qualifiers (`::`). Path qualifiers include `crate`, `$crate`, `super`, `self`, and in some cases, `Self`. Path segments must be lexically valid identifiers.

6.1.1. Simple paths

A simple path can optionally start with one of `crate`, `$crate`, `super`, or `self` (a path qualifier), and comprises one or more identifiers (simple path segments), separated by `::`. Simple paths are used in

- `use` declarations,
- Attributes,
- Macros, and
- Visibility specifiers.

For example,

```
use serde_derive::*;                    ①
use std::fmt::{Display, Formatter, Result};

#[derive(Serialize)]                    ②
#[serde(rename_all = "PascalCase")]
struct Car {
  num_wheels: u8,
  is_electric: bool,
}
```

① `use` directives. The curly brace `{}` syntax is explained later.

② Attributes.

6.1.2. Paths in expressions

Paths in expressions and patterns can optionally include generic arguments, e.g., as a path segment, `::<T>`. This is commonly known as "turbofish" syntax.

For instance,

```
#[derive(Debug)]
struct Gene<'a, T: Copy> {              ①
  dna: &'a [T],
}

impl<'a, T: Copy> Gene<'a, T> {         ②
  fn first(gene: &Self) -> Option<T> {
    if let &[f, ..] = gene.dna {        ③
      Some(f)
    } else {
      None
```

```
    }
  }
}

fn turbofish_demo<'a>() {
  let g = Gene::<'a, u8> {                ④
    dna: &[1, 2, 3]
  };
  println!("{g:?}");

  let f = Gene::<'a, u8>::first(&g);      ⑤
  println!("First element: {f:?}");
}
```

① The `struct` types, and generics, are explained in the Struct and Generics chapters, respectively.

② The implementations are also explained later in the Implementations chapter.

③ Likewise, refer to the later part of the book, e.g., on pattern matching and the `if let` expressions. If you are completely new to Rust, you do not have to understand all sample code at this point. All essential elements are clearly described throughout the book.

④ This is known as the struct literal, or the struct expression, and it constructs a new instance of a struct, e.g., `Gene` in this example. `Gene` is a generic type, and we use the turbofish syntax here to explicitly specify its generic arguments. Normally, Rust can infer generic arguments. Note that, unlike in many similar languages that support generics, Rust generally requires the leading `::` before the generic arguments. (See the next section, however.)

⑤ Another turbofish example. Just like the generic parameter declarations, generic arguments in expressions need to be specified in the order of lifetime, type, and const generic arguments.

6.1.3. Paths in types

Unlike the paths in general expressions, when type paths are used within type definitions, trait bounds, type parameter bounds, and other qualified paths, the leading `::` token can be omitted. For instance,

```
mod genetics {
  pub struct Gene<'a, T: Copy> {
    pub dna: &'a [T],
  }
  // etc...
}
```

```
fn splice(gene: genetics::Gene<u8>) {    ①
  // Do something with the gene...
}
```

① In the type context, as in this example, `Gene<u8>` is the same as `Gene::<u8>`.

Note that, in the expression and type context, the leading qualifier `Self` can also be used in paths, e.g., in addition to what is permitted in the simple paths.

6.2. Path Qualifiers

Paths can be denoted with a few different qualifiers, which affect their semantics.

6.2.1. Token `::`

Paths starting with `::` must be followed by the name of a crate in the extern prelude, and they are considered, and resolved, to be paths originating from that crate. Other identifiers in the path must resolve to items in that extern crate.

6.2.2. `crate`

The `crate` qualifier path segment resolves the path relative to the current crate.

6.2.3. `$crate`

The special qualifier, `$crate`, can only be used within macro transcribers, and it refers to a path to access the item from the top level of the crate where the macro is defined.

6.2.4. `self`

The path with the qualifier `self`, with a lowercase `s`, is resolved relative to the current module. Note that the identifier `self` is also used in the method context, as we describe later in the book.

6.2.5. `Self`

`Self` refers to the implementing type within traits and implementations. You can find some example uses of `Self`, and other path qualifiers, throughout this book.

6.2.6. `super`

The `super` qualifiers in a path resolve to their parent modules. Unlike other leading path qualifiers, they can be used after `self::`, and they can also be repeated, e.g., as in `super::super::super::xyz`.

For example,

```
fn crab() {
  println!("Just crab!");
}
```

```
mod sea {
  fn crab() {
    println!("Sea crab!");
  }
  pub mod world {                          ①
    pub fn crab() {
      super::super::crab();                ②
      self::super::crab();                 ③
    }
  }
}
```

① The `pub` visibility marker is discussed next.

② If we call `sea::world::crab()`, this statement will print out *Just crab!*.

③ This will print out *Sea crab!*. In the current context, `self::super::xyz`, for instance, is the same as `super::xyz`.

6.3. Visibility

Items in Rust can be either public or private. Public items are accessible to the outside world, so to speak. Otherwise, private items are only accessible from the same module. Associated items in a `pub` trait are public by default. Likewise, enum variants in a `pub` enum are also public, and their visibility cannot be separately specified.

6.3.1. Scope-based visibility

In a publicly visible item, additional items can be made public using a few different scope-specific visibility markers, e.g., in addition to the plain `pub`. Note that, to be able to access an item, all of its parent items up to the given scope must all be visible as well.

`pub(crate)`	This makes an item visible within the current crate.
`pub(super)`	This makes an item visible to the parent module.
`pub(self)`	This makes an item visible to the current module, which holds true by default, without this visibility marker.
`pub(in PATH)`	This makes an item visible within the provided path, `PATH`. The `PATH` cannot be arbitrary, and it must refer to an ancestor module of the given item.

6.4. The `use` Declarations

The `use` declarations create local name bindings based on some other paths. They can be used in modules and blocks, and they are usually placed at the top. Note that, unlike in some other common programming languages, the `use` directives are not used to "import" external items. In Rust, `use` is mainly used to shorten the name, or path, required to refer to an external item. That is, in general, `use` is not always required. A `use` declaration starts with the keyword `use`, and it is followed by some form of path specifications. We will not include the exact syntax, but here are some examples:

```
use a::b::c;                                    ①
```

① `a::b::c` can now be referred to as `c`.

```
use a::{b, c::d};                               ①
```

① This declaration is essentially the same as two declarations, `use a::b;` and `use a::c::d;`. And hence, `a::b` and `a::c::d` can now be referred to as `b` and `d`, respectively.

```
use a::b::{self, c};                    ①
```

① `a::b` and `a::b::c` can now be referred to as `b` and `c`, respectively.

```
use a::{self as x, c as y};             ①
```

① `a` and `a::c` can now be referred to as `x` and `y`, respectively. The `as` aliasing is commonly used to avoid name conflicts.

```
use a::*;                               ①
```

① Wildcard `use` syntax. For example, `a::x`, `a::y`, and `a::z::w`, assuming that they are valid paths, can now be referred to as `x`, `y`, and `z::w`, respectively.

6.4.1. pub use

The `use` declaration items are normally private to the containing module, like all other items. One can use the `pub use` syntax to make certain items public in the specified paths. The `pub use` declaration is syntactically the public version of the `use` declaration, but it provides the "export", or "re-export", mechanism from Rust modules, e.g., in addition to providing shorter names or aliases for longer paths. For example,

house.rs

```
mod dealer {
  pub use self::deck::deal_one;         ①
  mod deck {                            ②
    pub fn deal_one() -> i32 { 42 }     ③
  }
}
```

```
pub fn pub_use_demo() {
  // let uno = dealer::deck::deal_one(); ④
  let one = dealer::deal_one();          ⑤
  println!("dealt = {one}");
}
```

① "Re-exporting" the function item, `deal_one`.

② Note that the module, `dealer::deck`, is not visible from outside the `dealer` module.

③ On the other hand, the "re-exported" items still need to be visible. That is, you cannot make a private item public by declaring it in `pub use`.

④ The `deal_one` item cannot be accessed using the normal path since the item in the path, e.g., the `deck` module in this example, is not visible.

⑤ However, this "shortcut" works because of the `pub use` declaration.

6.4.2. The underscore alias

The `use` declaration can be used without binding names, e.g., by aliasing the path to an underscore (_). E.g.,

```
use self::me_mod::second::ThirdTraitor as _;
```

There are a few use cases. For example,

- To link an external crate without using any of its names,
- To "hide" a name from an otherwise imported set of names, and
- To import a trait, without importing its name, so that its methods can still be used.

6.5. Preludes

In general, the term "prelude" is often used in programming to refer to a collection of names that are automatically brought into scope in a program. Rust's standard library prelude includes names that are automatically imported, that is, unless the `no_std` attribute is used. These names are defined in, and/or re-exported from the `std::prelude::v1` module. They are always available in every module of any (non-`no_std`) Rust program.

These items can be referred to in a program without qualifications. We will go through a few of the types, such as `Option` and `Result`, and some of the traits, such as `Copy`, `Clone`, `Debug`, and (many) others, from the standard library prelude. As indicated, they are really part of the Rust language. even though they are defined in the "library".

6.5.1. Library preludes

Although they are not technically "preludes", many third-party libraries export names via the prelude convention.

A library crate may include a number of disparate modules, e.g., organized according to their functionalities, etc., and each of them may include a number of different items. In many cases, certain items may be considered more important, or more foundational, or more commonly used, etc., than others. The library authors often select a certain set of (crucial) items and put them in a module conventionally named `prelude`, e.g., using the `pub use` declarations.

Then, the users/clients of the library can conveniently (albeit explicitly) import all names from this special module, e.g., using one `use` directive. For example,

```
use killer_lib::prelude::*;
```

Builtin Attributes

The `no_std` Attribute

The standard library prelude, or the `std` crate, includes everything from the `std::prelude::v1` module. By default, they are automatically included in the crate root module, as if they are all a native part of the current module.

The `no_std` inner attribute may be applied at the crate level to prevent the `std` crate from being automatically added into scope. Furthermore, in `no_std` programs, Rust uses the names from the `core::prelude::v1` module instead. Note that `no_std` is often used on platforms where the standard library capabilities are not readily available. For example, on some platforms, there may not be heap memory available. Then, dynamic memory allocation may not be used on such platforms. On some platforms, they may lack file or network capabilities.

Builtin Attributes

The `no_implicit_prelude` Attribute

The `no_implicit_prelude` attribute works similarly to `no_std`. But, `no_implicit_prelude` can be applied at the crate level, as well as on a module, and they only affect the targeted module and their descendants. Note that Rust prelude can be classified into a few different categories, and `no_std` and `no_implicit_prelude` may work slightly differently. But, these details are beyond the scope of this book.

Chapter 7. Items

As stated, a Rust crate is made up of a (nested) set of items. Every crate has a single outermost item, i.e., an anonymous top-level module. All other items are organized within this module, and they have paths within the module tree of the crate. Some items form an implicit scope for the declaration of their subitems. That is, within a function or module, item declarations can be mixed with other statements, or control blocks, etc.

Rust defines a number of different items, as illustrated earlier. Let's *briefly* go over each of those items in this chapter.

7.1. Modules

Module items have been described earlier, in the context of the Rust program top-level structure.

7.2. Use Declarations

A `use` declaration creates one or more local name bindings synonymous with some other path. Use items have been described in the previous chapter, where we go through the basic uses of names and paths in Rust programs.

7.3. Constant Values

A `const` item represents an explicitly typed constant value with the `'static` lifetime, which outlives all other lifetimes in a Rust program. Const items are evaluated at build time, and they are *inlined*. One implication of this optimization is that a `const` item may not be associated with a single memory address. This is further discussed later in the `const` Items chapter.

7.4. Static Values

A `static` item is similar to a `const` item, except that it represents a single specific memory location in the program. All static items also have the `'static` lifetime. More details are provided later in the `static` Items chapter.

7.5. Type Aliases

Rust allows defining new names, or aliases, for an exiting type using the `type` alias declaration syntax. Type aliases are in a sense similar to the `use` items. Instead of providing short names for general item paths, type aliases provide a shorthand notation for other types, including generic types.

7.6. Struct Items

In Rust, the user can define custom types using `struct`, `union`, and `enum`. Structs are a generalization of the builtin tuple types. A struct type can include zero or more named or unnamed fields. For example,

```
struct Point {                          ①
  x: f32,
  y: f32,
  z; f32,
}
```

① A struct type with three named fields, `x`, `y`, and `z`, each of which is declared to be of the `f32` type.

We go through all essential elements of Rust's `struct` in some detail later in the `struct` Types chapter.

7.7. Union Items

The `union` declaration uses essentially the same syntax as the `struct` declaration, except that unions can have one and only one field. Rust's `union` is similar to that of the C programming language. Reading from unions is only allowed in unsafe code in Rust, and we will not further discuss unions in this book.

But, here's a quick example:

```
#[repr(C)]
union RiskyUnion {                          ①
  are_u: u32,
  press_f: f32,
}

fn union_demo() {
  let u = RiskyUnion { are_u: 1 };          ②
  let f = unsafe { u.are_u };               ③
  println!("{f}");
}
```

① A union includes one field, which can be interpreted in different types. In this example, it can be one of `u32` or `f32` types.

② An instance of a union type can be created using the syntax similar to the struct expression. In this example, we use the `u32` field. But, we can use the `f32` field as well.

③ Reading from a union field is inherently unsafe since we do not know the type of any given instance of the given union type. If we try to read the field of `u` as `f32`, e.g., as `u.press_f`, the program might panic. Or, even worse, we may end up with a completely wrong value, without any explicit errors.

7.8. Enum Items

Enums in Rust are much closer to Haskell's `data` types than C's enums. Rust's `enum` definition comprises a set of constructors, known as the *variants*. Enum values are often used in pattern matching, as we will further discuss later in the Enums chapter. A quick example:

```
enum Diet {                                  ①
  Keto,
  Paleo(u8, i128),
  Mediterranean(bool),
}
```

① This enum type consists of three variants.

7.9. Function Items

Functions in Rust are declared with the keyword `fn`, and they are rather similar to those found in other (imperative) programming languages, both syntactically and semantically. Functions can have zero or more input parameters, through which the caller passes arguments into the function, and they can return a value back to the caller on completion. Here's a simple example:

```
fn add_ten(arg: i32) -> i32 {
  arg + 10                                   ①
}
```

① The last expression in a function body, if any, becomes the return value from the function.

Function are further discussed in some detail later in its own chapter.

7.10. Trait Items

Rust's `trait` is comparable to Haskell's type class. It defines a behavior, or other characteristics, for a set of related types. That is, they are sort of the types of types, just like types are the types of values. A type becomes an *instance* of a trait by implementing the trait. Rust's traits and related concepts are explained in a few chapters later in the book, including the Traits chapter.

7.11. Implementation Items

An implementation, or `impl`, is an item that associates certain kinds of items, such as functions, with a specific type, called the *implementing type*. There are two types of implementations:

- Inherent implementations, and
- Trait implementations.

These are discussed in the Implementations chapter.

7.12. Associated Items

Associated items are the items declared in traits or defined in implementations. There are three kinds of items that can be associated with types (or, traits):

- Associated constants,
- Associated types (or, type aliases), and
- Associated functions and methods.

Inherent implementations can have associated constants and associated functions/methods, but not associated types. The Associated Items chapter later in the book provides some more details.

7.13. Extern Crate Declarations

An `extern crate` declaration specifies a dependency on an external crate. The external crate is then bound into the declaring scope. For example,

```
extern crate open_computer_vision;      ①
```

① The root module of the *open_computer_vision* crate can be referred to as `open_computer_vision` in this source file.

The `as` clause can be used to bind the imported crate to a different name.

```
extern crate open_computer_vision as opencv;
```

Although the `extern crate` declaration is still being used (and, it is not deprecated), it has very limited uses at this point. As can be seen from the above examples, their uses are now mostly replaced by the `use` declarations. We will not discuss the `extern crate` declarations any further in this book.

7.14. Extern Blocks

External blocks provide declarations of items that are not defined in the current crate. They are primarily used for importing libraries written in languages other than Rust. Two kinds of item declarations are allowed in `extern` blocks:

- Functions, and
- Static items.

One is permitted to call functions or access statics that are declared in `extern` blocks only in an unsafe context. Extern blocks are not further discussed.

Builtin Attributes

The `allow`, `warn`, `deny`, and `forbid` Attributes

In Rust, everything is integrated. For example, it's a lot easier to do unit testing in Rust than in many other (old-style) languages, in which they require extra setup and what not.

Linting is another example that is "natively" integrated into Rust's common tooling. Traditionally, linting (and, code formatting) has been a more important part of programming with dynamically typed languages like JavaScript due to their weaker compiler support (relatively speaking). But, this trend has been changing recently.

The Rust compiler will issue *warnings* if your code does not conform to certain predefined rules. For example, if you declare a variable and not use it, it will warn you, e.g., not because it itself is necessarily wrong but because it can potentially hide more serious errors.

The general *problem* with lint check is not that you do not agree with their general default rules but that, most of the time, your code is work in progress. As far as unused variables are concerned, for example, we can use certain naming conventions to stop the compiler from issuing unnecessary warnings, as stated earlier.

In general, however, you can selectively disable or enable particular lint rules using the attributes, `warn`, `deny`, or `forbid` as well as `allow`, any of which can be used either as an inner or outer attribute. These attributes can override the default, or previously set, values, if any, or the values set in the outer scope, etc., except for the ones set by `forbid`.

For any lint rule `rule`,

- `allow(rule)` overrides the check for `rule` so that violations will be ignored.
- `warn(rule)` warns about violations of `rule` but continues compilation.
- `deny(rule)` signals an error after encountering a violation of `rule`.
- `forbid(rule)` is the same as `deny(rule)`, but further changes are not allowed for this particular `rule`.

For example, *rustc* allows unsafe code by default. We can make it warn by using the *warn* attribute.

```
#![warn(unsafe_code)]

union UnsafeUnion {
  i: i32,
  f: f32,
}
fn union_demo() {
  let u = UnsafeUnion { i: 100 };
  let f = unsafe { u.f };                ①
  println!("{f}");
}
```

① This unsafe code will now issue a warning. If you are new to Rust, note that *cargo check* is your best friend during development, which essentially runs the compiler without producing a build output (which takes time). You can even try *cargo fix*, which is new as of Rust *1.69*.

As another example, disabling warnings from unused code can be rather useful during development.

```
#![allow(dead_code)]                          ①

enum Diet {                                   ②
  Vegan,
  Carnivore,
}
impl Diet {
  fn use_olive_oil() -> bool {                ③
    true
  }
}
```

① This is especially useful in the early stage of development.

② The unused enum item `Diet` does not issue a warning under this setting.

③ Likewise, the unused associated function (*unused*, at the moment) does not issue the *dead_code* warning.

You can also use the `allow`, `warn`, `deny`, or `forbid` settings for the lint settings as a command line flag to *rustc*.

```
rustc -W rule     Warn about rule
rustc -A rule     Allow rule
rustc -D rule     Deny rule
rustc -F rule     Forbid rule (deny rule and forbid override)
```

One can also use *clippy* for further lint checks, e.g., via *cargo clippy* or *cargo clippy --fix*, also new in *1.69*, or otherwise for other helpful feedback regarding your code, or your coding styles, in general.

Chapter 8. Rust Type System

Rust has a *static* and *strong* type system, which is largely influenced by ML-style languages like Haskell. The type system is, in fact, *the foundation* of the Rust programming language, and it is explained throughout this book. This chapter is included here as a quick introduction for the people who are new to Rust.

Overview

Here are a few salient features of the Rust type system:

- Rust supports traits, which can be viewed as the types of types.
- Rust's traits also play the role of interfaces or abstract classes in other languages.
- Some programming languages have the concepts of value types and reference types. Rust types are divided into copy types and move types, among others.
- Rust supports separate value and reference variables.
- Rust has no concept of `null` references (or, `nil` or `None`, etc.). All references should point to valid values in Rust.
- Rust supports both mutable and immutable variables.
- On the other hand, Rust does not support immutable types.
- All expressions in Rust are categorized into value expressions and place expressions, and they are evaluated either in the value or place contexts.
- Rust does not support Java/C# style OOP. In particular, Rust does not support type inheritance.

As indicated, we will elaborate on these points in the pertinent contexts throughout the book. But, let's briefly go over a few essential elements first.

Types and traits

Most programming languages, if not all, support the dual concepts of values (or, "objects") and types. Rust has three layers, values, types, and traits. This is one of the most important aspects of the Rust type system, which has broad implications when programming in Rust. This is further described in the rest of this chapter, to the benefit of the people who have no prior experience with the languages like Haskell which have similar type systems.

8.1. Traits as Type Classes

A type divides all values into two sets, one including the values which belong to the given type, and the other including those which do not. Likewise, a trait divides all types into two sets, one including the types which belong to the given trait, and the other including those which do not.

In other words, a type is a set of values. Likewise, a trait in Rust corresponds to a type class, which is a set of types. This is a very useful way to look at traits. Clearly, this is not the only viewpoint. A more common way of looking at traits as interfaces as in other programming languages is also a very useful way to view traits in Rust. In fact, these two different ways to understand traits are complementary, rather than alternative, views. Sometimes, one view is more useful than the other. One thing we need to emphasize, in the current context, is that interfaces in languages like Java, C#, and Go are types. On the other hand, traits are not types, as we just stated. This distinction has a practical importance.

One can use traits like types in Rust in two different contexts. An `impl` trait can be used like a (static) interface type, whereas a `dyn` trait can be used like a (dynamic) interface type. Both of these constructs will be briefly described later in a chapter titled Dynamic Dispatch (although `impl` traits are purely static constructs).

8.2. Copy vs Move

All types in Rust are either `Copy` or Move types. All types that implement the `Copy` trait are "Copy types". All other types that do not implement the `Copy` trait are "Move types". This is a dichotomy.

This idea is very *unique* to Rust. In fact, it is more so than any other concepts in Rust, including ownership and borrowing. If you are new to Rust, and if you don't have a firm grasp of this fundamental idea, then you will likely find learning and programming in Rust rather painful. Although it is a cliche, you will feel like the compiler is always yelling at you. You will feel like you always have to "fight the compiler". For example, some of the most common errors that the new Rust programmers tend to run into are "borrow after move", "borrow of moved value", or things of that sort.

Move semantics was originally introduced in C++, and it is now adopted by a few other languages such as C#. Let's briefly take a look at what it is.

The original, and still the most important, use case of move semantics in C++ is when you are returning a value object from functions (including constructors). Traditionally, *value* meant *copying* in imperative programming. In this particular use case, however, this is, or can be potentially, rather inefficient, especially if the return value happens to be "big".

What happens is, you have one value in a function scope, and when you return this value, you make a copy so that the caller can use this value, and then when the function is removed from the call stack, the original value is destroyed. Clearly, this is an inefficient way of doing things. We (almost always) end up making an "unnecessary" copy. "Moving" the originally value from the function scope to the outside caller's world is rather natural in this context since the callee function no longer needs it. What "moving" really means is actually an implementation detail, however. How C++ does it in particular situations, e.g., in terms of storage of these values in memory, is not very relevant to us.

In fact, as you can quickly realize, even if you have never used C++ or C# before, move semantics is an *optimization*. Other than the reasons for performance, or memory efficiency, *move* is an (almost) completely unnecessary construct in programming. Then came Rust. ☺ As indicated, the dichotomy of move vs copy in Rust is a fundamental part of the language. Every time you make an assignment, every time you use a pattern-matching expression, every time you call a function with arguments or you return a value from a function, etc., you will have to be *mindful* of what kind of types you are dealing with. As a matter of fact, when you design and write a program, when you create a new type, when you implement a function or method, you will need to think about this. Is it gonna be Copy or Move? We are just talking about value semantics here. But, on top of this, you will need to think about reference semantics as well.

(BTW, this concept does exist in C++, and the types that cannot, or should not, be copied are called the "move only" types, which correspond to the Move types in Rust. We will see some examples in a later chapter.)

8.2.1. The `Copy` trait

```
pub trait Copy: Clone { }
```

All types in Rust, builtin or custom, are Move types by default unless they implement the marker trait `std::marker::Copy`, defined in the Prelude. The values of a `Copy` type (e.g., a type that implements `Copy`) changes its move semantics to that of copy. `Copy` can only be implemented for types whose fields are all `Copy`. A type cannot implement both `Copy` and `Drop` traits at the same time.

```
#[derive(Debug, Clone, Copy)]
struct Point2D { x: i32, y: i32 }
```

8.3. Clone vs Non-Clone

All `Copy` types are *cloneable*. That is, by requirements, a `Copy` type has to be a `Clone` type. On the other hand, Move types are divided into `Clone` types and non-Clone types.

8.3.1. The `Clone` trait

The prelude trait `std::clone::Clone` defines the `clone` method, which is used for producing a copy of a value. `Clone` is a supertrait of `Copy`.

```
pub trait Clone: Sized {
  fn clone(&self) -> Self;
}
```

`Clone` is derivable for `struct` and `enum`, as long as their fields are all cloneable. For example,

```
#[derive(Debug, Clone)]
struct Qubit {
  phi: f32,
  psi: f32,
}
```

```
fn clone_demo() {
  let q1 = Qubit { phi: 0., psi: 3.14, };
  let q2 = q1.clone();
  println!("q1 = {q1:?}; q2 = {q2:?}");
}
```

8.4. Default vs No-Default

There are types that have default values, and there are types that do not have default values. More specifically, types that implement the `Default` trait have default values. On the flip side, for types that do not implement `Default`, we cannot rely on them having default values. Again, this is a dichotomy.

8.4.1. The `Default` trait

The `std::default::Default` trait defines one required function.

```
pub trait Default: Sized {
  fn default() -> Self;
}
```

8.5. Sized vs DST

All types in Rust belong to either `Sized` or Unsized. All types that implement the `Sized` trait are "Sized types". All other types that do not implement the `Sized` trait are "Unsized types" (or, dynamically sized types, or DSTs for short). Most of the types in Rust are `Sized` types.

In fact, `Sized` is a supertrait of `Clone`, which is in turn a supertrait of `Copy`, and hence all Copy types, as well as Cloneable types, are Sized types.

Only the values of a Sized type can be allocated on stack memory. To put it differently, values of Sized types can be stored either on stack or heap, whereas values of Unsized types can only use heap memory. Rust's basic types are all `Sized`, as are the vast majority of other builtin and user-defined types, with the exception of some (collection-oriented) types like `String` and `Vec`. They are dynamically sized, and their contents are always heap-allocated.

8.5.1. The `Sized` trait

The `std::marker::Sized` is one of the most fundamental traits in Rust. It is a marker trait, and it is always implemented automatically by the compiler when certain conditions are met. In fact, this trait cannot be explicitly implemented by user-defined types, either manually or through derive.

As stated, `Sized` refers to types whose sizes are known at *compile time*. That is, values of a `Sized` type have constant sizes, and they may be allocated on stack. Note that dynamic trait objects, `dyn TRAIT`, cannot be created for `Sized` traits. Rust's dynamic dispatch is discussed later in the book.

8.6. Deref - DerefMut Types

All types in Rust are either value types or pointer types. All types that specifically implement the `Deref` or `DerefMut` traits are pointer types. All other types that do not implement either of these traits are value types. For a value type `T`, there is generally a corresponding pointer type `&T`. Values of a pointer type can be "dereferenced", e.g., using the unary dereference operator, `*`. `Deref` is a supertrait of `DerefMut`. Values of a `DerefMut` type can be used with mutable references. In addition to overloading the dereference `*` operator, the `std::ops::Deref` and `std::ops::DerefMut` traits are also used in method resolution and deref coercions (which we do not discuss in this book).

8.6.1. The `Deref` trait

```
pub trait Deref {
  type Target: ?Sized;                    ①
  fn deref(&self) -> &Self::Target;
}
```

① `Target` is an associated type of `Deref`. We briefly discuss differences between generic traits and traits with associated types later in the book.

8.6.2. The `DerefMut` trait

```
pub trait DerefMut: Deref {
  fn deref_mut(&mut self) -> &mut Self::Target;
}
```

8.6.3. Pointer types

In Rust, immutable and mutable references, e.g., `&` and `&mut`, are pointer types. In unsafe Rust, (C-style) raw pointers can also be used. All standard library smart pointers are (specifically) pointer types. As indicated, any type that implements the `Deref` trait is a pointer type. And, on the flip side, a type that is not intended to be used like a pointer should not implement `Deref`.

8.7. Drop Types

All Move types are either `Drop` types or non-Drop types. `Copy` types cannot be `Drop` types. More specifically, all Move types that implement the `Drop` trait are "Drop types". All other Move types that do not implement this trait are "non-Drop types". Types of the values that would require cleanup after their use should be Drop types.

8.7.1. The `Drop` trait

The `std::ops::Drop` trait provides a destructor method, `drop`. The destructors are called when the values of a `Drop` type go out of scope, or they are destroyed otherwise, e.g., through an explicit `std::mem::drop` function call.

```
pub trait Drop {
  fn drop(&mut self);
}
```

The `drop` method is used to clean up the (non-memory) resources after their use, similar to the way instance destructors are typically used in C++. Note that this method is called by Rust, and you are not allowed to call this method directly. Attempting to do so will raise a compile error.

Builtin Attributes

The `derive` Attribute

The Rust compiler, or other implementations, may be able to provide basic implementations for some special traits. These traits are known as derivable traits. If the basic default implementation is sufficient for your specific type, you can use that implementation by declaring it so using the `derive` attribute. Note that you are not required to use the provided implementation of a derivable trait. It is a choice. You can still choose to explicitly implement the trait for a given type.

For example,

```
#[derive{Debug, Clone, Copy}]
struct Rocket(f64);
```

Chapter 9. Primitive Types

Rust supports most of the common primitive types found in (virtually) all other programming languages:

Boolean The `bool` type (`true` or `false`).

Numeric Integer and floating point types.

Textual Character and string types.

In addition, Rust includes builtin sequence, or composite, types like arrays, slices, and tuples as well as somewhat special `unit` and `never` types.

9.1. The Boolean Type

The boolean type `bool` is the usual primitive data type in Rust that can take on one of two values, `true` and `false`. Values of the `bool` type occupy 1 byte. All primitive types are defined in the language prelude.

```
pub const TRUTH: bool = true;
pub const FAKED: bool = false;
```

Like all primitives, the `bool` type implements the following core traits:

- `Sized`,
- `Clone`,
- `Copy`,
- `Send`, and
- `Sync`.

9.2. Numeric Types

9.2.1. Integer types

The unsigned integer types consist of:

`u8`	1 byte, 0 ~ 2^8-1 (Also used as a "byte" type in Rust)
`u16`	2 bytes, 0 ~ 2^16-1
`u32`	4 bytes, 0 ~ 2^32-1
`u64`	8 bytes, 0 ~ 2^64-1
`u128`	16 bytes, 0 ~ 2^128-1

The signed integer types consist of:

`i8`	1 byte, -2^8 ~ 2^7-1
`i16`	2 bytes, -2^16 ~ 2^15-1
`i32`	4 bytes, -2^32 ~ 2^32-1
`i64`	8 bytes, -2^64 ~ 2^63-1
`i128`	16 bytes, -2^128 ~ 2^127-1

9.2.2. Machine-dependent integer types

In addition, Rust supports two additional integral types, `usize` and `isize`, which have the same number of bits as the platform's pointer type (often known as the "word"). `usize` and `isize` are unsigned and signed types, respectively, and they are typically either 32 bit or 64 bit. `usize` can represent every memory address in

the process, and it is the type of an array index in Rust, for example. However, the (theoretical) maximum size of a value or array is the maximum value of `isize` (e.g., to ensure proper pointer arithmetic, etc.).

9.2.3. Floating-point types

The IEEE 754-2008 "binary32" and "binary64" floating-point types are represented by `f32` and `f64` in Rust, respectively.

`f32` 4 bytes, single precision

`f64` 8 bytes, double precision

Note that, unlike many (strongly typed) programming languages that still allow automatic "wider conversions", Rust does not permit implicit conversions even between similar numerical types. For example,

```
fn float_demo() {
  let a: f32 = 1.0;                    ①
  let b: f64 = 2.0;                    ②
  // let sum = a + b;                  ③
  let sum = (a as f64) + b;            ④
  println!("a + b = {}", sum);
}
```

① As we indicate in the Lexical Analysis chapter, the types of floating point numbers generally inferred to be `f64`, unless explicitly annotated, etc.

② Hence, this type annotation `f64` would have been redundant, in general.

③ The "narrower" type, `f32`, would have been automatically converted to `f64` in some other programming languages. In Rust, however, this results in an error.

④ Explicit type casting is needed here.

The `as` operator

The `as` operator can be used to convert between (compatible) primitive type values. In general, Rust supports various forms of type casting and conversions. Values of different (but, related) types can also be "coerced" into each other in certain situations, e.g., between a value and its reference, etc.

Furthermore, Rust provides a rather flexible type conversion framework, which can be used between just about any types, builtin or user-defined, e.g., using the `From` and `Into`, and other related, traits. We do not discuss type conversions any further in this book.

9.3. Characters and Strings

The types `char` and `str` hold textual data. A value of type `char` is a Unicode scalar value, represented as a 32-bit unsigned word in the ranges from `0x0000` to `0xD7FF` or from `0xE000` to `0x10FFFF`. A value of type `str` is represented in the same way as `[u8]` (a slice of *bytes*). However, the data in `str` must be a valid UTF-8 string. Since `str` is a dynamically sized type, it can only be used/referenced through a pointer type, such as `&str`. As we indicated earlier, the type of string literals is `&str`. For example,

```
fn str_demo() {
  let (x, y) = ("Hello", "Crab");         ①
  println!("{x}, {y}!");                  ②
}
```

① The type of both `x` and `y` is `&str`. We describe the related type `String` later.

② This will output *Hello, Crab!*. In this example, we use essentially the `format!` macro syntax to concat two strings. In case of string literals, the `concat!` macro can also be used. E,g., `concat!("Hello", ", ", "Crab", "!")`.

9.4. The Unit Type

The unit type `()` includes one valid value `()`, which represents "no valid value" ☺, or *absence of any valid value*. The use of the symbol `()` for the unit value is based on the fact that tuples, along with structs, etc., generally represent "multiplicative types". The *empty tuple* is hence the unit value (like `0` for addition and `1` for multiplication). The unit is, however, not a compound type, although it uses the *same syntax* as tuples in Rust. It is a *primitive type*. For example,

```
fn point_of_no_return() -> () {          ①
  println!("A crab never returns!");
}
```

① This function does not return any meaningful value, and hence its return value is `()`, whose type is `()`.

9.5. The Never Type

The never type `!` is a special type with no valid values, commonly used in functional programming. It represents the result of the computation that never returns, e.g., due to an infinite loop. Or, a function that simply terminates a program is a function that *never returns*, as far as the program is concerned. We will see a few uses of `!` in this book, but it's primarily there for formality.

```
fn point_of_no_u_turn() -> ! {           ①
  panic!("A crab never U-turns!");
}
```

① This function simply terminates the program through the `panic!` macro, and hence it *never returns*.

Builtin Attributes

The `feature` Attribute

When you use a nightly build of Rust, which includes many unstable features, you can specifically opt in to use any of those unstable features using the `feature` attribute. For example,

```
#![feature(is_some_and)]                    ①

fn is_some_and_demo() {
  let (x1, x2, x3) = (
    Ok::<i32, ()>(10),
    Ok::<i32, ()>(-10),
    Err::<i32, _>("Huh?"),
  );
  assert_eq!(x1.is_ok_and(|x| x > 0), true);
  assert_eq!(x2.is_ok_and(|x| x > 0), false);
  assert_eq!(x3.is_ok_and(|x| x > 0), false);
}
```

① The `feature` attribute with the "is_some_and" feature, which enables `is_some_and`, and other related methods such as the one shown here, `is_ok_and`, on types like `Option<T>` and `Result<T, E>`. This feature can only be enabled when you use a nightly build of Rust.

This is called the feature flags. Note that there are numerous "features", which come and go, in the nightly builds, and these features, and even their names, are, by definition, *unstable*.

Chapter 10. Tuples and Sequence Types

Rust supports the common sequence types such as arrays and slices. We also briefly describe tuples in this chapter, which are Rust's other builtin compound types.

10.1. The Array Types

An array is a sequence of `N` elements of type `T`, with a constant `N` (of type `usize`). The (fixed-size) array type is written as `[T; N]`. The dynamic-size array type, `Vec<T>`, is described in the next chapter. Vectors are heap-allocated, whereas the array values are (by default) stack-allocated. In Rust, when an array is created and initialized, the elements of the array are also all initialized. Furthermore, in Safe Rust, access to arrays is always bounds-checked.

Otherwise, Rust's arrays are rather similar to those found in other programming languages. For example,

```
fn array_demo() {
  let arr1 = [1, 2, 3, 5];                ①
  println!("arr1 = {arr1:?}");
  println!("arr1[1] = {}", arr1[1]);      ②
  // println!("arr1[5] = {}", arr1[5]);   ③

  let mut arr2 = [2u8, 4, 8, 10];         ④
  println!("arr2 = {arr2:?}");
  arr2[2] = 88;                           ⑤
  println!("arr2[2] = {}", arr2[2]);
}
```

① The type of this array is `[i32; 4]`.

② You can access an element of the array using the index, or subscript, operator. Note that we do not go through all syntax of Rust in this book. Being familiar with some commonly used, and presumably more basic, syntax of C-style languages should really be considered a prerequisite to reading this book.

③ Attempting to access an invalid index will cause a compile or runtime error, depending on whether the `#[deny(unconditional_panic)]` annotation is set or not (which is on by default).

④ As of the current version of Rust (*1.69*), there is no easy way to specify only the type of the elements but not the size. That is, `[u8; 4]` can be used as a valid type annotation, as well as `[_, 4]`. But, the discard symbol `_` cannot be used in place of the size. This syntax is sort of a workaround. We force the type of the elements to be `u8` without having to explicitly specify the array size.

⑤ When an array variable is declared as mutable, we can update the values of the elements of the array, e.g., again using the familiar element access syntax.

Rust's array types come with a number of convenience methods, including `get` and `get_mut`, etc. These two functions, for example, do not cause compile time errors (or, runtime errors). Instead, they return an `Option` value. For instance,

```
fn array_get() {
  let arr = ["Hell", "World", "Rust"];
  for i in [0, 2, 4] {
    if let Some(x) = arr.get(i) {        ①
      println!("arr[{i}] == {x}");
    } else {
      println!("No value at index {i}");
    }
  }
}
```

① The `if let` expression is explained later in the book, if you haven't seen, or used, it before. This statement, inside the `for` loop, will end up printing *arr[0] == Hell*, *arr[2] == Rust* and *No value at index 4* in three separate lines.

10.2. Array Expressions

Array expressions, sometimes called array literals, create new arrays. There are two different syntax. First, we can sequentially list all elements in the array, separated by commas, e.g., `[e1, e2, e3]`.

Alternatively, an array can be created using the repeat element and length operands separated by a semicolon (`;`), e.g., `[e; length]`, where is `e` is the element to be repeated. The length operand should be a constant expression of the `usize` type. When the length `>= 1`, the repeat operand should either be a `Copy` type, or it should be a constant value or a path to a constant item.

For example,

```
fn array_expressions() {
  let a1 = ["Feliz", "Ferris"];          ①
  let a2 = ["Crab"; 4];                  ②
  println!("{a1:?}");
  println!("{a2:?}");                    ③
}
```

① The first form of creating an array. This array expression, on the right hand side, is sometimes called the *array literal*.

② The alternative form of creating a new array. This array expressions will create a four-element array with the same (repeat) element, `"Crab"`.

③ This will print *["Crab", "Crab", "Crab", "Crab"]*.

10.3. The Slice Types

A slice represents a "view" into a sequence of values, and they are dynamically sized, unlike arrays. The slice type with an element type `T` is written as `[T]`, but similar to the `&str` type (which is really a slice type), they are generally used as a pointer type:

- `&[T]`: A "shared slice", or just "slice". A slice does not own the data it points to. It borrows it.
- `&mut[T]`: A "mutable slice". A mutable reference to the underlying data.
- `Box<[T]>`: A "boxed slice".

Similar to arrays, safety of access to slices are ensured by Rust. That is, elements of a slice are always initialized, and their access is always bounds-checked in safe methods and operators. A slice can be created based off an array or a `Vec`. For instance,

```
fn slice_from_array() {
  let arr = [5, 10, 20, 42];
  let s1 = &arr as &[i32];              ①
  println!("s1 = {s1:?}");
  let s2 = &arr[1..=2];                 ②
  println!("s2 = {s2:?}");
}
```

① A slice can be created from an array reference by casting it to a slice type. The type of `&arr` is `&[i32; 4]`, and the type of `s1` is `&[i32]`, and it is a "view" to the entire array.

② A slice can also be created using the range syntax. The slice `s2` in this example is based on the array elements, `arr[1]` and `arr[2]`. One can take a full slice as well, e.g., using the full range syntax, `&arr[..]`.

Mutable slices can also be created from arrays or vectors.

```
fn slice_from_array_2() {
  let mut arr = [5, 10, 20, 42];          ①
  let m1 = &mut arr as &mut [i32];        ②
  println!("m1 = {m1:?}");
  m1[1] = 11;                             ③
  println!("m1 = {m1:?}");
  println!("arr = {arr:?}");              ④
}
```

① A mutable array. (Or, more precisely, a mutable variable pointing to an array.)

② A mutable slice, `m1`, of type `&mut [i32]` based on the mutable array.

③ We can update the *content* of the mutable slice using the index notation.

④ The content of the underlying array is also modified. This `println!` statement will output *arr = [5, 11, 20, 42]*.

Like arrays, the slice types also come with a number of builtin convenience methods, such as `get` and `get_mut`, etc. The interested readers can look them up on the relevant API docs. In general, Rust has an extensive set of methods defined for all builtin types (and, for those in the Standard Prelude, etc.), and going through these APIs is well beyond the scope of this book.

Values of all "builtin" sequence types like arrays, slices, and vectors can be *iterated*. That is, these types all implement the `IntoIterator` trait.

```
fn slice_iteration() {
  let list = vec!["Do", "Re", "Mi", "Fa"];
  for v in &list {                        ①
    println!("{v}");
  }
```

```
    let slice = &list[..];                          ②
    for (i, s) in slice.iter().enumerate() {        ③
      println!("{s}{}", "~".repeat(i + 1));
    }
}
```

① The `for` expression, and the iterator traits, are described later. Note that we iterate over the reference of the vector `list`.

② We take a slice of the vector.

③ The `enumerate` method returns a series of pairs of the index and the corresponding value from the iterator.

10.4. The Tuple Types

Tuples are Rust's builtin, light-weight, structural types, comprising one or more fields. They are often called the multiplicative types, or product types, because of the way they are composed of multiple types. These field types can all be different, and hence tuple types are generally heterogeneous, unlike arrays or vectors. (The term *product* comes from the "(outer) product set" in set theory.)

Unlike `struct` types, the fields of a tuple type cannot be assigned custom names. Instead, they are just named with indices, e.g., `0`, `1`, `2`, etc. Tuple fields can be accessed by either a tuple index expression or pattern matching. A tuple type has a number of fields equal to the length of the list of types. This number of fields determines the *arity* of the tuple. A tuple with `n` fields is called an *n-ary tuple*.

The syntax for a tuple type is a parenthesized, comma-separated list of types. For example, `(bool, i32, f32)` is a 3-ary tuple type with its field types, `bool`, `i32`, and `f32`. For 1-ary tuple types, the trailing comma is required, e.g., `(u8,)`. Values of tuple types are constructed using tuple expressions, which are sometimes called the tuple literals (although they are not technically literals).

10.5. Tuple Expressions

A tuple expression creates a new tuple value, with a fixed number of elements, or the tuple initializer operands, separated by commas. The number of initializer operands are called the *arity* of the tuple (as in unary, binary, ternary, etc.), corresponding to the arity of the tuple type. The `0`-ary tuple syntax, that is, a pair of empty parentheses `()`, is the unit.

For the `1`-ary tuple expression, a comma after the (only) initializer operand is required, e.g., to distinguish it from a parenthetical expression. Otherwise, the trailing commas are optional. This syntax reflects that of the tuple types.

For example,

```
fn tuple_expressions() {
  let t0 = ();                                ①
  let t1 = (true,);                           ②
  let t2 = ('O', 2u8,);                       ③
  let (name, title) = ("Ferris", "King Crab".to_string());
  let t3 = (name, "the", title);              ④
  println!("{t0:?}; {t1:?}; {t2:?}; {t3:?}");
}
```

① A unit tuple expression, which is not really a tuple. The type of `t0` is the unit type, `()`. The `let` binding is explained later in the book.

② A `1`-ary tuple. Its type is `(bool,)`.

③ A `2`-ary tuple, whose type is `(char, u8)`. The trailing comma is optional.

④ The type of `3`-ary tuple, `t3`, is `(&str, &str, String)` since the types of `name`, `"the"`, and `title` are `&str`, `&str`, and `String`, respectively.

10.6. Range Expressions (`..` and `..=`)

The `..` and `..=` operators will construct a value of one of the `std::ops::Range` variants, according to the following rules:

Range Expression: `start..end`

type: `std::ops::Range`
range: *start* ≤ *x* < *end*

RangeFrom Expression: `start..`

type: `std::ops::RangeFrom`
range: *start* ≤ *x*

RangeTo Expression: `..end`

type: `std::ops::RangeTo`
range: *x* < *end*

RangeFull Expression: `..`

type: `std::ops::RangeFull`
range: -

RangeInclusive Expression: `start..=end`

type: `std::ops::RangeInclusive`
range: *start* ≤ *x* ≤ *end*

RangeToInclusive Expression: `..=end`

type: `std::ops::RangeToInclusive`
range: *x* ≤ *end*

```
let r1 = 1..=5;
r1.for_each(|v| print!("{v}, "));
```

Chapter 11. Other Basic Types

Rust relies on a number of core types that are defined in the standard library. Although they are defined in the standard library, they are as important, and as often used, as the *primitive types*. Some other types like `Box<T>` are discussed in the later part of the book.

11.1. The `String` Struct

The Standard Prelude includes the `String` struct and `ToString` trait from the `std::string` module, and hence syntactically they are used just like builtin structs and traits. The `String` type represents the dynamic-sized, heap-allocated strings. All Rust strings are UTF-8-encoded, including the primitive `&str` type. Unlike `&str`, however, which is a borrowed reference, `String` has ownership over the *content* of the string. Rust comes with many builtin `String` methods such as `len`, `push`, `trim`, and `split`, etc. We do not include them in this book, however.

11.1.1. Constructors

The most common way to create a `String` is to call `String::from` (from the `std::convert::From` trait) with a string literal. For example,

```
fn string_from() {
  let mut king = String::from("King ");
  king.push_str("Crab");
  king.push('!');
  println!("{king}");                    ①
}
```

① Note that `String` implements `Display` while `&str` does not.

11.2. The `Option<T>` Enum

The `std::option::Option<T>` type represents an optional value, or, the presence or absence of a value. It includes two variants `Some<T>` and `None`, and both are exported from the Standard Prelude.

11.2.1. Trait implementations

The `Option<T>` enum implements the following standard library traits, among many others: `Clone`, `Debug`, `Default`, `Hash`, `From`, `FromIterator`, `IntoIterator`, `Ord`, `PartialEq`, `PartialOrd`, `Product`, `Sum`, `Copy`, and `Eq`. (Note that, for example, Options are Copy types. It is beyond the scope of this book, but as we briefly discuss in an earlier chapter on Rust's type system, the traits that a type implements dictate how the type can be used, and in fact, what that type *is*.)

11.2.2. Variants

The `Option<T>` enum includes two variants:

```
pub enum Option<T> {                    ①
  None,
  Some(T),
}
```

① Enum types are explained a bit later in the book, but one can think of an `enum` as a type consisting of a set of *constructors*. If you are coming from languages like Java or C#, or any imperative programming languages, the term constructor has subtly different connotations in Rust (and, in other ML-style languages). In case of `Options<T>`, `None` is a no argument constructor. It creates, or returns, a single value `None`. On the other hand, the variant `Some(T)` is a constructor that takes one argument of a (generic) type `T`.

For example,

```
fn option_demo() {
  let b1: Option<bool> = None;           ①
  let b2: Option<bool> = Some(true);     ②
  println!("b1 = {b1:?}; b2 = {b2:?}");
  assert_ne!(b1, b2);                    ③
}
```

① `None` is a constructor, which "creates" sort of a *singleton* value, of the `Option<_>` type.

② `Some(true)` create a value of `Option<bool>` type.

③ `b1` and `b2` have the same type but different values.

11.2.3. Implementations

Values of `Option<T>` are commonly used in `match` expressions or in other related pattern-based expressions. Besides, the `Option<T>` type implements a number of methods, many of which are commonly, and routinely, used in Rust programming. Although it is well outside the realm of what this book focuses on, some of the more commonly used methods are

- `is_some(&self) -> bool`,
- `is_none(&self) -> bool`,
- `expect(self, msg: &str) -> T`,
- `unwrap(self) -> T`,
- `unwrap_or(self, default: T) -> T`,
- `unwrap_or_default(self) -> T: Default`, and
- `unwrap_or_else<F>(self, f: F) -> T where F: FnOnce() -> T`.

11.3. The `Result<T, E>` Enum

The `std::result::Result<Type, Error>` type includes two variants, `OK<Type>` and `Err<Error>`, which represent either success or failure, respectively. Although it is similar to the `Either<Left, Right>` type in Haskell, for instance, `Result<T,E>` is primarily, and almost exclusively, used as a return type from a function that can succeed or fail. In case of failure, the error message is often included in the `Err<E>` value. Like `Option<T>`, both of its variants are exported from the Standard Prelude.

11.3.1. Trait implementations

The `Result<T,E>` enum implements the following standard library traits, among others: `Clone`, `Debug`, `Default`, `Hash`, `From`, `FromIterator`, `IntoIterator`, `Ord`, `PartialEq`, `PartialOrd`, `Product`, `Sum`, `Copy`, and `Eq`.

11.3.2. Variants

```
pub enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

For example,

```
fn result_demo() {
  let r1: Result<u8, &str> = Ok(42);
  let r2: Result<u8, &str> = Err("Help!");
  assert_ne!(r1, r2);                    ①
}
```

① `Ok(42)` and `Err("Help!")` look so different, but they have the same type, e.g., `Result<u8, &str>` in this example. Rust's enums are often called the union types for this reason, e.g., as in the set unions. (Not to be confused with *C* unions. We do not discuss Rust's `union` types in this book.)

11.3.3. Implementations

Similar to `Option<T>`, values of `Result<T, E>` are commonly used in the `match` expressions, or in other pattern-matching contexts. And, likewise, the `Result<T, E>` type also implements a number of methods, many of which are integral part of Rust programming. Some of the more important methods are

- `is_ok(&self) -> bool`,
- `is_err(&self) -> bool`,
- `expect(self, msg: &str) -> T where E: Debug`,
- `unwrap(self) -> T where E: Debug`,
- `unwrap_or(self, default: T) -> T`,
- `unwrap_or_else<F>(self, f: F) -> T where F: FnOnce() -> T`,
- `unwrap_or_default(self) -> T: Default`,
- `ok(self) -> Option<T>`,
- `err(self) -> Option<E>`,
- `and<U>(self, res: Result<U,E>) -> Result<U,E>`,
- `or<F>(self, res: Result<T,F>) -> Result<T,F>`, and
- `map<U,F>(self, op: F) -> Result<U,E> where F: FnOnce(T) -> U`.

These `and`, `or`, and `map` methods have a number of variations. The readers are encouraged to look up these methods on Rust's official website. In Rust, programming well, and effectively, often means using methods like these well.

11.4. The `Vec<T>` Struct

Vectors, defined in the `std::vec` module, are another important standard library types that support heap-allocation of their values. They are dynamic-sized, and their *content* cannot be stored on stack memory. Similar to `String`, a `Vec` does not directly contain the data that it owns. Instead, it stores a pointer to the data (which is stored on the heap). In addition, a `Vec` struct stores the length of the data as well as the capacity of the current storage location. Note that these are all private fields and they are not accessible.

```
pub struct Vec<T> {
  /* private fields */
}
```

11.4.1. Constructors

`Vec` implements a few functions that are used as constructors.

```
const fn new() -> Vec<T>;                          ①
```

① The `new` function constructs a new, empty `Vec<T>`. Elements can be pushed into a vector, e.g., an instance of `Vec`, using various methods of `Vec`.

```
fn with_capacity(capacity: usize) -> Vec<T>;    ①
```

① This function creates a new, empty instance of `Vec<T>` with a minimum specified capacity. Note that the length of the constructed vector, either via `new` or `with_capacity`, is zero, regardless of its initial capacity. As described earlier, the `vec!` macro is also commonly used to construct a vector and initialize its elements in one go.

Chapter 12. Core Traits

The following traits are defined in the standard library, but they play essential roles in Rust, just like other standard library types, if not more. Types that implement these core traits, builtin or user-defined, get special treatment by the Rust compiler, as we briefly alluded earlier. Here's a quick legend.

A	Auto trait. These terms are explained later.
B	Blanket implementation.
D	Derivable via the `#[derive]` attribute.
M	Marker trait.
O	Operator overloading. More operator traits are listed later.
Prelude	Whether exported in the Standard Prelude or not.
Supertraits	Trait dependency.

This is just a quick table, for your future reference. You do not have to memorize this like learning the multiplication table.

The following traits should be considered the core of the core traits.

	A	B	M	O	D	Prelude	Supertraits
`Sized`			O			O	
`Clone`					O	O	`Sized`
`Copy`			O		O	O	`Clone`
`Drop`						O	

`Debug`					O	`std::fmt`	
`Display`						`std::fmt`	

The following traits are also rather commonly used across many different areas in Rust programming.

	A	B	M	O	D	Prelude	Supertraits
`Any`		O				`std::any`	
`Hash`					O	`std::hash`	
`Default`					O	O	`Sized`
`Deref`				*		`std::ops`	
`DerefMut`				*		`std::ops`	`Deref`
`Send`	O		O			O	
`Sync`	O		O			O	
`Unpin`	O		O			O	
`UnwindSafe`	O					`std::panic`	
`RefUnwindSafe`	O					`std::panic`	
`Error`						`std::error`	`Debug` + `Display`
`Termination`						`std::process`	

The following traits define conversion behavior.

	A	B	M	O	D	Prelude	Supertraits
`ToOwned`		O				O	
`AsRef<T>`						O	
`AsMut<T>`						O	

`From`		O				O	`Sized`
`Into`		O				O	`Sized`, (`From`)
`TryFrom`		O				O	`Sized`
`TryInto`		O				O	`Sized`, (`TryFrom`)
`FromStr`						`std::str`	`Sized`
`ToString`						`std::string`	(`Display`)
`Borrow<T>`		O				`std::borrow`	
`BorrowMut<T>`		O				`std::borrow`	`Borrow<T>`

Function traits. One cannot directly implement these traits (e.g., in safe Rust, at this point). They are, for example, automatically implemented for closures.

	A	B	M	O	D	Prelude	Supertraits
`FnOnce<T>`	_			`()`		O	
`FnMut<T>`	_			`()`		O	`FnOnce<T>`
`Fn<T>`	_			`()`		O	`FnMut<T>`

The following traits, including common comparison operators, are used for operator overloading, among other things.

	A	B	M	O	D	Prelude	Supertraits
`Index<T>`				`[]`		`std::ops`	
`IndexMut<T>`				`[]`		`std::ops`	`Index<T>`
`PartialEq`				`=`	O	O	
`PartialOrd`				`<`	O	O	`PartialEq<Rhs: ?Sized>`

`Eq`				`=`	O	O	`PartialEq<Self>`
`Ord`				`<`	O	O	`Eq` + `PartialOrd<Self>`

The following traits are used to define iterators.

	A	B	M	O	D	Prelude	Supertraits
`Iterator`						O	
`IntoIterator`						O	
`FromIterator<T>`						O	`Sized`
`ExactSize Iterator`						O	`Iterator`
`DoubleEnded Iterator`						O	`Iterator`
`Extend<T>`						O	

Async-related traits:

	A	B	M	O	D	Prelude	Supertraits
`Future`						`std::future`	
`IntoFuture`						`std::future`	

IO-related traits:

	A	B	M	O	D	Prelude	Supertraits
`Read`						`std::io`	
`Write`						`std::io`	

12.1. Auto Traits

A few Rust builtin traits are designated as *auto traits*, and they are specially treated by the compiler. If no explicit implementation or negative implementation is provided for any of the auto traits for a given type, then the compiler implements it automatically, if feasible, according to some predefined rules. The following are auto traits:

- `std::marker::Send`,
- `std::marker::Sync`,
- `std::marker::Unpin`,
- `std::panic::UnwindSafe`, and
- `std::panic::RefUnwindSafe`.

12.2. Marker Traits

The following are *marker traits*, and they do not define any associated items such as methods. As we explain in an earlier part of the book, the primary use of these traits is to classify types into different classes. Besides `Sized` and `Copy`, the rest three traits also play important roles in defining the characteristics of types, but we do not cover them in this book.

`Sized`	Types with a constant size known at compile time.
`Copy`	Types whose values can be duplicated simply by copying bits.
`Send`	Types that can be transferred across thread boundaries.
`Sync`	Types for which it is safe to share references between threads.
`Unpin`	Types that can be safely moved after being pinned.

12.3. Derivable Traits

The Rust compiler, or other implementations, may be able to provide basic implementations for some special traits. These traits are known as derivable traits. If the basic default implementation is sufficient for your specific type, you can use that implementation by declaring it so using the `derive` attribute. Note that you are not required to use the provided implementation of a derivable trait. The following traits are derivable traits:

- Comparison traits: `Eq`, `PartialEq`, `Ord`, and `PartialOrd`.
- `Clone`: To create `T` from `&T` via a copy.
- `Copy`: To give a type *copy semantics* instead of *move semantics*.
- `Hash`: To compute a hash from `&T`.
- `Default`: To create an empty instance of a data type.
- `Debug`: To format a value using the `{:?}` formatter.

12.4. Blanket Implementations

Rust automatically and generically implements a certain set of traits over all types, e.g., without trait bounds. These are known as the blanket implementations. For example, this will constitute blanket implementations for the trait `Trait`:

```
trait Trait { fn do_nothing(); }         ①
impl<T> Trait for T { fn do_nothing() {} }
```

① We use Rust's basic constructs like traits and impls throughout this book without precisely defining them first. The exact syntax for `trait` and `impl` is discussed later.

Here are some of the standard library traits that provide blanket implementations:

- `std::any::Any`,
- `std::borrow::Borrow<T>`,
- `std::borrow::BorrowMut<T>`,
- `std::borrow::ToOwned`,
- `std::convert::From<T>`,
- `std::convert::Into<U>`,
- `std::convert::TryFrom<U>`, and
- `std::convert::TryInto<U>`.

12.5. Type Conversion Traits

Readers should be familiar with a general pattern found in many programming language, in which certain core language features can be *customized* through various "hooks" built into the language. Use of the *dunder methods* in Python is one such example. Another most common example is the use of a particular method to convert an (arbitrary) object to string (e.g., `toString` method).

Many of these "hooks" are also built into Rust. We briefly discuss the operator overloading later in the book, for example. This mechanism is also utilized in type conversions. Traits like `From`, `Into`, `TryFrom`, and `TryInto` play crucial roles in this context. The readers are encouraged to refer to other references for more information. We show one example of using the `From` trait in the last chapter, in the context of error handling. Incidentally, if you implement `From` or `TryFrom` on a type, Rust automatically provides sensible implementations for `Into` or `TryInto`, respectively. They are marked in the table earlier with parentheses, in the supertraits column. The same comment applies to `ToString` vs `Display`.

Chapter 13. Formatting

The `std::fmt` module includes the language-level support for the `format!` extension and other related *types*, *traits*, and *macros*. As we briefly discuss earlier in the common macros chapter, the format strings can include a number of formatting arguments (`{}`), which can be either positional or name-based. These arguments can additionally include extra parameters for their value representations (e.g., decimal vs binary) and precision, and for other specifications such as width, fill, and alignment, etc.

13.1. Formatting Traits

13.1.1. The `Display` trait

The `std::fmt::Display` trait defines one method `fmt`, which is to be used for an empty format, `{}`.

```
pub trait Display {
  fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

For example,

```
pub struct FortyTwo {}
impl Display for FortyTwo {
  fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error> {
    42
  }
}
```

```
let age42 = FortyTwo {};
println!("I am {}", age42);
```

Implementing this trait for a type will automatically implement the `ToString` trait for the type, which defines the (ubiquitous) `to_string` method. It is generally a common practice to implement `Display` and not `ToString`.

13.1.2. The `Debug` trait

The `std::fmt::Debug` trait is rather similar to `Display`, in terms of its uses and what not, and it also defines one method `fmt` with exactly the same signature as `Display::fmt`. But, unlike `Display`, `Debug` is derivable, and in practice, this `fmt` method is rarely explicitly implemented, if ever.

```
pub trait Debug {
  fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

The `Debug::fmt` method is used for debug formats, `{:?}` and `{:#?}`. For instance,

```
#[derive(Debug)]                          ①
struct City {
  name: &'static str,
  pop: u64,
}
fn debug_trait_demo() {
  let ny = City {
    name: "New York",
    pop: 8_888_000,
  };
```

```
  println!("{ny:?}");
  println!("{ny:#?}");
}
```

① Pretty much every type implements `Debug` as a starter, that is, unless they implement `Display`, in which case `Debug` becomes "less important".

This will output something like this:

```
City { name: "New York", pop: 8888000 }
City {
    name: "New York",
    pop: 8888000,
}
```

13.1.3. The `Write` Trait

The `std::fmt::Write` trait defines the `write_str` required method, and it provides two other convenience methods.

```
pub trait Write {
  fn write_str(&mut self, s: &str) -> Result<(), Error>;
  fn write_char(&mut self, c: char) -> Result<(), Error>
  { ... }
  fn write_fmt(&mut self, args: Arguments<'_>) -> Result<(), Error>
  { ... }
}
```

This trait is specifically used for writing, or formatting, an argument into Unicode-accepting buffers or streams. In general, the `std::io::Write` trait can be used for broader use cases, such as when you need explicit flushing, etc.

13.2. Formatters

The `std::fmt` module includes traits, other than `Debug` and `Display`, that are commonly used for formatting (primarily) numeric outputs. They all define a single required method that is the same across all these traits:

```
fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
```

Here's a quick list.

Trait	Flags	Synopsis
`Debug`	`?`, `#?`	—
`Display`	—	—
`Binary`	`b`, `#b`	Formats its output as a number in binary.
`Octal`	`o`, `#o`	Formats its output as a number in octal.
`LowerHex`	`x`, `#x`	Formats its output as a number in hexadecimal, with `a` through `f` in lowercase.
`UpperHex`	`X`, `#X`	Formats its output as a number in hexadecimal, with `A` through `F` in uppercase.
`LowerExp`	`e`	Format its output in scientific notation with a lowercase `e`.
`UpperExp`	`E`	Format its output in scientific notation with an uppercase `E`.
`Pointer`	`p`	Format its output as a memory location, e.g., in hexadecimal.

Chapter 14. The Const Items

A `const` item in Rust is a (usually named) constant value which is not associated with a specific memory location in the program. Constants are essentially inlined wherever they are used, meaning that they are copied directly into the relevant context when used. References to the same constant are not necessarily guaranteed to refer to the same memory address.

14.1. Named Constants

Constants must be explicitly typed. The type must have a `'static` lifetime. Any references in the initializer must have `'static` lifetimes.

Constants may refer to the address of other constants, in which case the address will have elided lifetimes where applicable. Otherwise, in general, they will default to the `'static` lifetime. Regardless, the address of constant items may not be stable, and they should not be relied on in the program.

Here are a few examples of constant values:

```
const MyTruth: bool = true;
const MyPi: f32 = 3.14;
const MyGreeting: &str = "Hello!";
```

14.2. Unnamed Constants

Constants can also be declared with a discard variable (_). They are called the *unnamed constants*. Unnamed constants are evaluated at compile time, just like their named counterparts, and that is often the sole purpose of their uses. That is, unnamed constants are mainly used to check the code validity at build time.

For example,

```
const _: i32 = 1 + 2;                    ①

struct S {
  f1: bool,
  f2: i128,
}

const _: S = S {                         ②
  f1: true,
  f2: 1111111111,
};

const _: () = {                          ③
  let a = 123;
  let b = 456;
  let _c = a + b;
};
```

① Not very useful, but legal.

② Checks if we can construct an instance of S at compile time.

③ In fact, we can execute essentially *any code*, using a block expression, at compile time. Whether it is a good practice or not is a different question, however. ☺

Chapter 15. The Static Items

A `static` declaration introduces a static name to a program, which have the `'static` lifetime, just like the `let` binding introduces local names. The `'static` lifetime outlives all other lifetimes in a Rust program. A `static` item declaration requires an explicit type specification, similar to `const` items, and it is followed by a `static` initializer, which also needs to be a constant expression evaluated at compile time. Static initializers may refer to other static or constant items, and constant functions. The reverse, however, does not hold true. That is, constant initializers cannot refer to static values.

Unlike `const` items, static items are associated with their own memory locations in the program. All references to the same static item refer to the same memory location. Static items can be mutable, but reading from, and writing to, mutable static items can only be done in unsafe code. In general, use of constants should be preferred over static items, if possible, e.g., unless interior mutability is required or large amounts of data need to be stored, etc.

Here are a few examples of using const and static values. Traits, associated items, and implementations are explained later in the book.

```
trait Diet {
  const FOOD: &'static str;
  fn eat(&self);
}
impl Diet for i32 {
  const FOOD: &'static str = "Grapes";
  fn eat(&self) {
    static MEAL: &'static str = i32::FOOD;      ①
    println!("I only eat {MEAL}");
  }
}
```

```
static F: &'static str = "Olive oil";
fn statics_demo() {
  println!("{F}");
  static D1: i32 = 42;
  D1.eat();
}
```

① A `static` item can refer to `const` items, `const` values or `const` function calls.

Builtin Attributes

The `inline` Attribute

The `inline` attributes can be applied to functions or closures, and they suggest to the compiler that the attributed items should be inlined, if possible. Inlining means that a copy of the function/closure definition should be directly placed in each calling code rather than making a separate call expression.

```
fn fun_1() -> i32 {
  ({ #[inline] |x, y| x + y })(1, 2)     ①
}
#[inline(always)]                        ②
fn fun_2() -> f64 { std::f64::consts::PI }
#[inline(never)]                         ③
fn fun_3(x: bool, y: bool, z: bool) -> bool { (x || y) && z }
```

① `#[inline]` suggests performing an inline expansion.

② `#[inline(always)]` suggests that inlining should *always* be performed

③ `#[inline(never)]` suggests that inlining should *never* be performed.

Chapter 16. Variables

A *variable* in Rust is, or corresponds to, a location in memory on a stack frame. They can be local variables, or function parameters, or sometimes anonymous temporary memory locations.

16.1. Local Variables

A local variable holds a value on stack memory. Hence their lifetime is tied to that of the particular stack frame which the variable belongs to. Local variables come into existence when their memory is allocated on a stack frame, and their memory is de-allocated when the frame is removed from the call stack.

There are two kinds of local variables. The variables that are used to directly keep their values, and those that are used to refer to other values. The former is often called the value variables and the other is generally called the pointer or reference variables. Reference variables hold values (on stack) that are the addresses of storage locations (on stack or heap) that hold the referenced values. Since a reference variable can refer to another reference variable, this "pointing" relationship can continue indefinitely. Once the value dereferenced is not an address, the chaining ends. Note that values of certain types cannot be stored on stack, e.g., because they are dynamically sized, that is, because their sizes are unknown at compile time. (If you are new to this whole business of stack vs heap, etc., then this is a basic knowledge in order to be able to become proficient in programming in Rust. For example, the stack can only hold values whose sizes are known at compile time. Otherwise, you cannot "stack" variables on tope of each other, so to speak.)

The size of a memory address is fixed, and it is always known (e.g., on a particular platform). They are typically either 32 bits or 64 bits. Therefore, we have no problem storing reference variables on the stack. On the other hand, as stated, only the values of `Sized` types can be stored on the stack.

Variables can be either *mutable* or *immutable* in Rust (but, not both), as we discuss a bit later in this chapter. Hence, local variables can be classified into four categories, value mutable, value immutable, reference mutable, and reference immutable. As indicated, unsized types like `String` or `Vec` cannot use value variables, mutable or immutable, because the variables live on the stack whereas the values of these types are always stored on the heap.

16.2. Function Parameters and Return Values

Function parameters, in effect, define local variables within the context of the function body. Hence most of the things we discuss with respect to local variables in this chapter, and throughout the book, apply also to function parameters.

Although it can be slightly different in certain situations, function return values involve the same or similar rules to those of local variables. Sometimes, function return values may be temporarily stored in an anonymous memory location.

16.3. Scoping

Scopes play an important role in all programming languages. This is especially so in Rust. Scopes are used by the compiler to keep track of ownership, borrowing, and lifetimes. Different constructs in Rust have different scoping rules, but as far as the local variable is concerned, its scope is defined from its definition, e.g., when its memory is allocated on the stack frame, to its destruction, e.g., when the frame associated with the variable is removed from the stack.

Rust is a lexically scoped language like many other C-style languages. This means that, among other things, variables' scopes are determined purely by the program source code at compile time.

16.4. RAII

RAII is one of those words that sound like one thing but turn out to be another. RAII stands for *Resource Acquisition Is Initialization*. This is one of the innovations that C++ brought on top of C. The concept is rather simple. A value is created using a constructor, in which any necessary resources are acquired, including memory, and when this value goes out of scope, its destructor is called, in which any acquired resources are freed, including memory.

In principle, this should rid the world of all resource leaks and memory-related problems. In reality, however, that is not the case. The main culprit is the pointers. Resources are often shared, and RAII alone does not bring the world peace.

16.5. Rust's Ownership Model

Every value in a Rust program, which can be referenced with a local variable, has one, and only one, *owner*. This ownership concept is essentially based on the C++-style RAII model. But, it has rather different nuances, and it has very different implications.

First, RAII is generally tied to a single owner variable in Rust. Rust enforces the RAII rule based on the lifetime of the owner variable (which may be the same as, or smaller than, that of the value it holds). When the owner goes out of scope, its destructor is called if one is defined, and the memory is de-allocated, and all other resources owned by that value is freed. Second, as stated, all (local) variables live on stack, being associated with particular stack frames with finite lifetimes. None of the variables are "permanent". None of the values are "permanent", except for the exceptions of `const` and `static` values. This is one of most difficult things to learn for people who have background in memory-managed programming languages. For example, in languages like Java, JavaScript, Python, etc., all values are heap-allocated, except for some *minor* exceptions like primitive type values. In Rust, it's right on your face. *Nothing lives forever.* Well, almost nothing. There is an

escape hatch, as we will further discuss later in the book, e.g., in the context of smart pointers and what not. But, regardless, this is one of the things that a new Rust programmer will have to confront, and learn, sooner or later.

Third, it's a bit less significant, but still there is obviously a problem if only one variable (designated as the owner) can use any given value. This is where *borrowing* comes in. Rust is ultimately a reference-based programming language. It is a lot easier to manipulate reference variables than value variables.

16.6. Borrowing

As just stated, references are more commonly used in Rust than values themselves. Taking a reference, from a value or through another reference, is called *borrowing* in Rust. For example, instead of passing values *by value* (`T`) to a function, values can be passed *by reference* (`&T`), without the overhead, or complications, of copies or moves of the values. (Obviously, this is not specific to Rust.) The Rust compiler uses what is called the borrow checker to ensure that, at compile time, references always point to valid values. That is, as long as there are references pointing to a value, the value will not be destroyed. On the flip side, no references pointing to a given value can outlive that value.

16.6.1. Shared vs mutable borrows

To ensure this, Rust requires the programs to follow certain rules.

- A value (`v`) can have any number of immutable references (`&v`) at any given time, often called the immutable or shared borrows, but
- It can have no more than one mutable reference (`&mut v`), called the mutable or exclusive borrow.

As long as there is an active mutable borrow, there cannot be any other shared borrows, and the original value cannot be used.

Rust's ownership and borrowing model is one of the most common topics that a beginning Rust programmer first encounters, and virtually all beginner's books cover this topic thoroughly. If this concept is not clear to you, we recommend the readers to consult other references, including "the book".

16.7. Lifetimes

The same comment applies to lifetimes as well. It's such a basic, and yet somewhat difficult to understand, concept to new comers to Rust. We recommend the readers to consult other references if you are completely new to Rust.

A *lifetime* is a rather unique construct in Rust. The borrow checker uses lifetimes to ensure that all borrows are valid. It is based on, and a generalization of, the lexical scopes which most programmers should be familiar with, and which we briefly mentioned a bit earlier in the chapter.

In general, a value's lifetime begins when it is created, that is, when its memory is allocated, and it ends when it is destroyed. When we use references, or borrows, however, it is not always clear where a reference's lifetime begin or where it ends. For example, a borrow variable may be passed into a function and returned from a function, etc. In such cases, which is almost always the case when references are involved, the borrow checker requires explicit *lifetime annotations* (except when lifetimes can be elided), e.g., to determine how long each reference should be valid. The lifetime parameters start with apostrophes (`'`), and they are normally used as generic parameters, or as part of reference type declarations. For instance,

```
fn lifetime_demo_1<'a>() -> &'a str {     ①
  let x = "hello";
  let y: &'a str = &x;
  y
}
```

① `<'a>` is a generic parameter, and the return type `&'a str` in the function signature indicates that the return value is of the `&str` type (e.g., a reference) and it lives as long as the function itself. The function implementation itself is a bit convoluted since the string literal has the `'static` lifetime.

As this example illustrates, using lifetimes often requires generics, as we further discuss in the next chapter, and throughout the book. Items like functions, methods, structs, traits, and impls can be declared with lifetime parameters.

16.7.1. Lifetime elision

Some lifetime patterns are rather common that the borrow checker will allow you to omit them. This is called *elision*. We will not go into the detailed rules of lifetime elision in this book, but it should be noted that these elision rules exist because the compiler can infer a sensible default choice.

16.8. Shadowing

In many different programming languages, variables of the same name can be declared, and re-declared, in different nested scopes. The later-declared variable, e.g., in an inner scope, is said to *shadow* the earlier-declared variable with the same name, e.g., in the outer scope. For example,

```
fn shadowing_demo() {
  let x = 21;
  if true {
    let x = 42;                    ①
    println!("{x}");               ②
  }
  println!("{x}");                 ③
}
```

① This new `x` shadows the outer `x`.

② This prints *42*.

③ This prints *21* since `x` refers to the `x` in the outer/function scope.

Rust goes one step further. One can *shadow* variables with the same names even in the same scope. This is, in fact, a rather common practice in functional programming, in which you are generally not allowed to *mutate* variables or values. Using a similar example,

```
fn rust_shadowing_demo() {
  let x = 21;
  println!("{x}");                    ①
  let x = 42;                        ②
  println!("{x}");                    ③
}
```

① This prints *21*. Note that we no longer need to use the value `21` after this point.

② The second `let` binding shadows the first `x`. In the imperative style, we would declare the original `x` as mutable, e.g., `let mut x = 21` and then we would mutate `x` to a new value `42`. Shadowing helps avoid *unnecessary mutations*.

③ This prints *42* since `x` now refers to the second `x`.

One thing to note is that, unlike in the scope-based shadowing, when a variable is shadowed by another variable in the same scope, there is no way to refer back to that original variable/value, unless the value has other references.

Another thing to note is that shadowing does not change the scope or lifetime of the shadowed variable. That is, for instance in this example, the first `x` (and, the value `21`) will be de-allocated when it reaches the end of the function block, and not when it is shadowed. Clearly, this can have some practical implications when borrowing is involved. But, we will not go into any further details in this book.

16.9. Place Expressions vs Value Expressions

Expressions in Rust, which we discuss a bit later in the book, belong to one of the following two categories:

- *Place expressions*, or
- *Value expressions*.

These two categories roughly correspond to *lvalues* and *rvalues*, respectively, in C++. Other programming languages that support pointers, e.g., such as Go, also support similar concepts.

A place expression in Rust is an expression that represents a *memory location*. A value expression is an expression that represents an *actual value*. It is often important to distinguish one from the other when evaluating expressions. They are also closely related to the concepts of value vs reference, as explained in the beginning of this chapter.

We will not elaborate on this topic further in this book, but, for completeness, paths referring to the following items or expressions are place expressions:

- Local variables,
- Static variables,
- Dereferences, `*expr`,
- Array indexing expressions, `expr[idx]`, and
- Field references, `expr.f`.

All other expressions are value expressions in Rust. Grouped, or parenthesized, expressions belong to the same category as their operands.

16.10. Implicit Borrows

In certain situations, some expressions will be automatically treated as borrows, and hence they become effectively place expressions. They are called *implicit borrows*. One common example of implicit borrows is comparison expressions, in which moving their operands (or, even just copying) would not generally make much sense, e.g., just to compare their values. For reference, reference semantics may be used through implicit borrows in the following expressions:

- Operand of dereference (`*`),
- Left operand of array indexing (`[]`),
- Left operand of field access (`.`),
- Left operand of compound assignments,
- Left operand in function or method call expressions, and
- Operands in binary comparison expressions.

For instance,

```
fn implicit_borrow_demo() {
  let arr = [1, 2, 3];
  let _a = arr[0];                    ①
  let _a = arr.index(0);              ②
  let _a = (&arr).index(0);           ③
}
```

① The `[]` operator is applied to a value `arr`. But, there is no copy/move involved.

② `arr[0]` is the same as calling the `Index::index(&self, 0)` method on `arr`.

③ In both cases, implicit borrow is taken. They are equivalent to this expression, which is in turn equivalent to `(&arr)[0]`.

Chapter 17. Generics

In strongly typed languages, situations often arise where certain sets of types or functions have almost the same declarations and/or definitions except for some other types or parameters involved.

For example, Rust includes an (infinite) series of array types, e.g., `[i32; 4]`, `[i32; 5]`, ... `[String; 10]`, `[String; 100]`, etc., etc., which are distinct but all related, in some sense. Generally speaking, arrays are an example of parametrized types, e.g., `[T; N]` with two parameters, `T` and `N`, denoting a type and an integer, respectively.

Rust supports generics, just like virtually all modern statically and strongly typed languages. Rust's generics is influenced by Haskell's parametrized type system, and it is rather similar to the modern C++'s templates (with concepts) and C#'s generics (with type constraints), among others.

If you are familiar with any of these languages, Rust's generics is pretty much the same, except for some syntactic differences. In particular, Rust uses what is called the "trait bounds" to define, or constrain, a set of related types when declaring generic types, generic implementations, or generic functions. In addition, the concept of "lifetime bounds" is rather unique to Rust.

17.1. Generic Parameters

Generics can be used with functions, type aliases, and types such as struct, enums, and unions, as well as traits and impls. Since Rust 1.65, associated types can now be generic as well. The generic parameters are enclosed in angular brackets (`<...>`), separated by commas, and the general syntax puts them between the generic item name and its definition. For impls, however, they come directly after the keyword `impl`.

In Rust, there are three kinds of generic parameters, and they need to be included in this order:

- Lifetime parameters,
- Type parameters, and
- Const parameters.

Most generic systems found in other programming languages primarily make use of the type parameters. The lifetime parameters are rather unique to Rust, and their primary use is to specify the lifetime bounds, as we describe shortly. We next take a look at the (relatively new) `const` parameters in Rust's generics.

17.2. Const Parameters

Rust's `const` generic parameter syntax allows items to be parametrized over constant values, e.g., similar to the way array types are parametrized by array sizes. Syntactically, the `const` keyword before an identifier, followed by a bool or integral type, specifies a `const` parameter. One can use all integer types, including `char`, and the `bool` type for `const` parameter types. That is, `u8`, `u16`, `u32`, `u64`, `u128`, `usize`, `i8`, `i16`, `i32`, `i64`, `i128`, `isize`, `char`, and `bool` are allowed. For instance, here's a simple generic struct type that is parameterized both over a type parameter `T` and a constant value parameter `N`:

```
#[derive(Debug)]
struct AnArray<'a, T: Copy,                ①
  const N: usize>([&'a T; N]);             ②
```

① This is a *tuple struct,* which is discussed later in the book. For illustration, this generic struct `AnArray` is just a simple wrapper over an array type.

② Note the syntax `const N: usize`. Despite syntactic similarity, the colon `:` here does not signify a trait bound.

Here's a simple example use of the `AnArray` type:

```
trait TheArray {                                    ①
  type Item: Copy;
  fn get(&self, i: usize) -> Option<Self::Item>;
}
impl<'a, T: Copy, const N: usize> TheArray for AnArray<'a, T, N> {
  type Item = T;                                    ②
  fn get(&self, i: usize) -> Option<Self::Item> {
    if i < N {
      Some(*self.0[i])
    } else {
      None
    }
  }
}
fn const_generics_demo() {
  let a = AnArray::<'static, char, 3>([&'a', &'b', &'c']);
  for i in [2, 10] {                                ③
    if let Some(e) = a.get(i) {                     ④
      println!("Element at {i} = {e}");
    } else {
      println!("No element at index {i}");
    }
  }
}
```

① A trait with an associated type.

② As we will discuss later, although they are closely related, generic traits and traits with associated types are used slightly differently.

③ This `for` loop will end up printing *Element at 2 = c* and *No element at index 10*.

④ The `if let PATTERN` expression is explained later in the book.

17.3. Trait Bounds

In general, generic definitions may be applicable only to a certain limited *set* of types. The trait bounds (and, lifetime bounds) are used to restrict which types (and, lifetimes) can be used as their generic parameters.

In fact, *the* primary use of traits in Rust is to use them as trait bounds. The fact that a trait represents a *set* of types, as mentioned earlier, plays an important role in this context. On the flip side, (custom) traits are less frequently used in Rust, e.g., outside this particular use case, compared to (user-defined) interfaces in other programming languages. In those languages, interfaces are types, which can be used in both static and dynamic contexts, and it is generally considered a best practice to use a (broader) interface type rather than a (specific) concrete type. That is not generally the case in Rust, however.

17.3.1. The `where` clause

Bounds can be provided, e.g., using a `where` clause, on any type and lifetime parameters. The `where` clause can also be used to specify bounds on types that are not type parameters. In addition, the `for` keyword can be used to introduce higher-ranked lifetimes. Here are examples of a generic type and a generic function which use trait bounds:

```
#[derive(Debug, Clone)]
struct Cell<'a, T> where T: Clone + 'a { ①
  nucleus: T,
  mitochondria: &'a T,
}
```

① The type `T` must implement `Clone`. And, all references in `T` must outlive the generic lifetime `'a`, which is tied to `Cell`.

```
fn replicate<'a, T>(cell: Cell<'a, T>) -> Cell<'a, T>
  where T: Clone + 'a {                    ①
  Cell {
    nucleus: cell.nucleus.clone(),
    mitochondria: cell.mitochondria,
  }
}
```

① Ditto. Note that since the function `replicate` uses the type `Cell`, its trait bound must be the same as, or more restrictive than, that of `Cell`.

Here's another example which uses a trait bound on an associated type:

```
trait LiveForever<'a>
  where Self::LifeForm: 'a + Copy + Display + From<&'a
Self::LifeForm> {
  type LifeForm;                           ①
  fn make_life(creature: &'a Self::LifeForm) -> Self::LifeForm;
}
```

① The trait bound for the associated type `LifeForm` is specified with its parent trait, `LiveForever`, in this example.

The trait bound syntax

As can be seen from these examples, a trait/lifetime bound is specified with the `where` keyword, the target generic parameter, and a colon (`:`), followed by one or more traits and/or lifetimes. In case there is more than one, these traits and lifetimes are separated by plus symbols (`+`).

When there is more than one generic type parameter that needs to be specified in a single `where` clause, they are separated by commas (`,`).

17.3.2. The shorthand notations

In some common situations, a shorter form syntax can be used in liu of the explicit `where` clause. For example,

```
enum Food<T: Clone + Display> {          ①
  Vegetable(T),
}
```

① This declaration is the same as

```
enum Food<T>
  where T: Clone + Display,              ①
{ Vegetable(T), }
```

① The trailing comma is optional.

Likewise,

```
trait Human {}
trait Baby: Human {                      ①
  fn smile(&self);
}
```

① The syntax `trait Baby: Human ...`, denoting that `Human` is a supertrait of `Baby`, is equivalent to

```
trait Baby where Self: Human {
  fn smile(&self);
}
```

In case of associated types,

```
trait Train<'a> {
  type Car: 'a + Drop;                    ①
  fn add_car(&mut self, car: &'a Self::Car);
}
```

① This trait bound syntax over an associated type is the same as

```
trait Train<'a>
  where Self::Car: 'a + Drop,
{
  type Car;
  fn add_car(&mut self, car: &'a Self::Car);
}
```

17.3.3. The `?Sized` bound

As indicated earlier, the `Sized` trait is special in that it is automatically included in all generic trait bounds even when no trait bound is explicitly specified. The `?Sized` notation can be used to relax the implicit `Sized` trait bound for the generic type parameters and associated types.

```
enum Either<'a, T: ?Sized> {
  Heaven,
  Hell(&'a T),                            ①
}
```

① The variant `Hell` can contain a reference of any type, including unsized dynamic types.

17.4. Lifetime Bounds

A lifetime bound, `'a`, can be applied to types or to other lifetimes.

- When a lifetime bound is applied to a type, e.g., `T: 'a`, it requires that
  - All references in `T` must outlive the lifetime `'a`, and
- When a lifetime bound is applied to another lifetime, e.g., `'lifetime : 'a`, it requires that
  - The lifetime `'lifetime` must last at least as long as the lifetime bound `'a`. This is usually read as `'lifetime` outlives `'a`. This means that a reference of `&'lifetime T` must be valid as long as the reference of `&'a T` to the same value is valid, for any type `T` (that satisfies other trait bounds).

For example,

```
#[derive(Debug)]
struct RefVal<'a, T>(&'a T)                ①
  where T: 'a + PartialOrd;                ②
```

① As indicated, the reason why we use a lifetime *generic* parameter in the first place is to use it in lifetime bounds.

② The type `T` must implement the `PartialOrd<T>` trait, and the reference contained in this `RefVal<T>` must outlive the lifetime `'a`.

Here's another example, using this tuple struct, `RefVal`,

```
fn bigger<'a, 'b, 'c, T>(
  x: &'a RefVal<'a, T>,
  y: &'b RefVal<'b, T>,
) -> &'c RefVal<'c, T>                     ①
```

```
where
  'a: 'c,                                    ②
  'b: 'c,
  T: 'c + PartialOrd,
{
  if x.0 >= y.0 { x } else { y }
}
```

① A somewhat convoluted example to illustrate use of the lifetime bounds.

② This lifetime bound requires that the lifetime of `x` must span the entire lifetime of the returned reference value.

```
fn lifetime_bound_demo() {
  let (x, y) = (1, 2);                       ①
  let (rx, ry) = (RefVal(&x), RefVal(&y));

  let max = bigger(&rx, &ry);                ②
  println!("{max:?}");
}
```

① The scope of the variables, `x` and `y`, and hence the lifetimes of `&x` and `&y`, start from the next line, and they extend to the end of this function body block.

② The type of `max` is `&RefVal(i32)`. Note that `&rx` and `&ry`, and hence `&x` and `&y`, outlive `max`.

17.5. Higher-Ranked Trait Bounds

In certain situations, a lifetime bound may need to be applied to a single trait, e.g., within a trait bound, rather than to the entire trait bound. In such a case, the lifetime bound is called the higher-ranked lifetime bound, and it uses a special syntax using the `for` keyword.

For example,

```
fn apply_fn<F1, F2>(f1: F1, f2: F2)
where
  F1: for<'a> Fn(&'a mut i32),          ①
  F2: for<'b> Fn(&'b i32),              ②
{
  let mut x = 3;
  f1(&mut x);
  f2(&x);
}
```

① The lifetime parameter `'a` only applies to the following trait `Fn(&'a mut i32)`. Syntactically, this trait bound is equivalent to `for<'a> F1: Fn(&'a mut i32)`.

② Likewise, this trait bound `F2: for<'b> Fn(&'b i32)` is the same as `for<'b> F2: Fn(&'b i32)`.

```
fn increment<'a>(x: &'a mut i32) {
  *x += 1;
}
fn print<'a>(x: &'a i32) {
  println!("value = {}", *x);
}

fn hrtb_demo() {                        ①
  apply_fn(increment, print);
}
```

① A demo of using this `apply_fn` function. Note that this example is for illustration purposes only, and in fact the lifetime parameters in this particular example are all optional.

Chapter 18. Functions

18.1. Functions

Functions may be declared either as an item or as an associated item, e.g., in a `trait` or in an `impl` block. A function consists of

- The keyword `fn`,
- A name, which is required,
- A set of zero or more parameters in parentheses,
- An optional return type after `->`, and
- A function body block.

The body block may be omitted for a function associated with a `trait`, and, in such a case, the function declaration terminates with a semicolon. For example,

```
fn add_each_other(x: i32, y: i32) -> i32 {
  x + y
}

trait Multiplier {
  fn mul_42(&self, x: i64) -> i64;
}
```

18.1.1. Function parameters

As we describe a bit earlier in the book, function input parameters are also (local) variables. As with `let` bindings, function parameters are irrefutable patterns. Any pattern that is valid in a `let` binding is also valid as a function parameter.

In case of associated function, the first parameter can be one of the following six `self` parameter forms:

- `self`,
- `mut self`.
- `&self`,
- `&mut self`,
- `&'lifetime self`, and
- `&'lifetime mut self`.

Note that `mut self` is essentially the same as `self`. In case of `extern` block functions, the last parameter can be a variadic parameter (`...`), which can optionally be preceded by a name, e.g., `the_rest: ...`.

18.1.2. Function body block

The function body is conceptually a block expression. It uses the function arguments as locally declared variables, and the value of the block essentially becomes the output which the function returns to its caller on completion.

18.1.3. Function types

Function types are denoted by the `fn` keyword, followed by a parameter list, and optionally `->` and the return type, if any. For example, the `add_each_other` function above has a type,

- `fn(i32, i32) -> i32`,

whereas the type of the `Multiplier::mul_42` method is

- `fn(&Multiplier::Self, i64) -> i64`.

18.1.4. Function as a value

A function, when it is referred to, e.g., by its name, yields a *first-class value* of the corresponding (implicit) function type. When this value is called, it evaluates to a direct call to the function. For example,

```
fn add_21(x: i32) -> i32 {
  x + 21
}

fn function_demo() {
  let f = add_21;                          ①
  let result = f(21);                      ②
  println!("result = {result}");
}
```

① The variable `f` is bound to a function of the type `fn(i32) -> i32`. Note that functions are Copy types. (See the next subsection.)

② This value, which `f` refers to, can be directly called.

18.1.5. Trait implementations

All functions implement all auto traits as well as the following three function traits:

- `std::ops::FnOnce`,
- `std::ops::FnMut`, and
- `std::ops::Fn`.

In addition, they all implement `Clone`, `Copy`, `Debug`, `Pointer`, `Hash`, `PartialEq`, `Eq`, `PartialOrd`, and `Ord`, among others.

18.2. The `const` Functions

Functions declared with the `const` keyword are `const` functions, and they are evaluated at compile time. (And, they cannot be `async` or `extern`.) The implementation of a `const` function should only include constant expressions, and `const` functions can be called from within other `const` contexts.

```
const A: i32 = 32;                          ①
const B: i64 = 64;
const C: i64 = cfun();                      ②

const fn cfun() -> i64 {
  A as i64 + B + 48                         ③
}
```

① Const items are explained earlier in the book.

② A `const` function can be called in the `const` context.

③ A `const` function can only include constant values, and it is evaluated at compile time.

18.3. The `async` Functions

Functions qualified with the `async` keyword are `async` functions. For example,

```
async fn async_function() { }
```

An `async` function does not execute its body when it is called. Instead, it captures its arguments into a `std::future::Future`, and it returns that `Future`. When, and only when, that future is polled, it will execute the function's body.

For example,

```
async fn afun() -> i32 {                    ①
  42
}

async fn async_function_demo() {            ②
  let x = afun();                           ③
  let y = x.await;                          ④
  assert_eq!(42, y);
}
```

① Calling an `async` function returns a `Future` despite its apparent form. We use a trivial function for illustration here, but in general, `async` functions are used for IO- or CPU- intensive tasks.

② We can only use `await` expressions in `async` functions or `async` blocks.

③ Calling `afun()` returns a type of `impl Future<Output = i32>`.

④ Then, we can `await` on that `Future`, which returns the computed value when the task completes. In this example, the value and type of `y` are `42` and `i32`, respectively.

We will briefly go over the `async` block expressions later. As indicated, however, we do not go into async-await programming in any depth in this book.

One thing to note is that although Rust *does* do provide this basic construct, it does not provide any `async` runtime support. Currently, you will have to rely on the third-party crates like *tokio* or *async-std* for async programming (which are, btw, fantastic libraries). This might, and *should*, change in the near future, however. The currently used rationale behind not providing the async runtime as an (optional) part of the standard library is rather weak, in our opinion.

Chapter 19. Closures

Rust supports anonymous functions, also known as lambda functions or lambda expressions. They are called the closures in Rust. Rust's closures are *closures*, that is, they can capture variables from the outside scope. A closure expression in Rust defines a closure type and evaluates to a value of that type, which can be called in place, assigned to a variable, or passed in to a function as an argument, among other things.

19.1. Closure Expressions

The syntax for a closure expression is

- A comma-separated list of closure parameters, enclosed in a pair of vertical bars (`|...|`),
  - Each of which is a pattern, and
  - Each of which can be followed by an optional type annotation,
- An optional arrow token (`->`) and a return type, and then
- An expression, called the closure body operand.

The closure expression can be optionally preceded by the `move` keyword, whose semantics is explained later. When the return type is explicitly specified, the closure body must be a block.

For example,

```
fn closure_demo_1() {
  let c1 = |x, y| x + y;                ①
  let v1 = c1(1, 2);                    ②
  assert_eq!(v1, (|a, b| a + b)(1, 2));  ③
```

```
    use std::f32::consts::PI;
    let c2 = |x: f32| -> f32 { x * PI };    ④
    let v2 = c2(2.0);
    println!("value = {v2}");
}
```

① The type of `x` and `y` of the closure `c1` is inferred to be `i32` based on its use in the next line.

② A closure can be called like a function.

③ Calling the closure, `|a, b| a + b`, in place.

④ Alternatively, we can explicitly specify the types of the closure parameters and their return values, if any. Note that, in this case, (i) we need to use a block expression (`{}`) since the return type is specified, and (ii) the type annotations are not necessary since we use the `f32` version of `PI` in the closure declaration.

A closure expression maps a list of given arguments to the value prescribed by the closure body expression. Like the function parameters, closure parameters are also irrefutable patterns. Unlike the function definitions, however, type annotations are generally optional for closures since the types can be more easily inferred from the context.

In many modern programming languages, functions, either named or unnamed, are generally closures. In Rust, however, the regular functions do not capture their environment. Only closures do. Therefore, there are some subtle differences in usages and best practices when it comes to Rust's closures.

For example, one of the main advantages of using lambda functions over named functions in many other languages is that lambda functions do not need be separately declared before their use. In those languages, declaring a lambda and assigning it to a variable for later use is often considered a bad practice since the named functions can be more suitable for that purpose.

In contrast, the choice of the (named) function vs (anonymous) closure in Rust is primarily determined by the following two factors:

- Whether it's going to be called more than once, and
- Whether it needs to capture the environment.

When a closure expression is defined, the Rust compiler infers how it captures each variable from its environment that is mentioned in the closure's body. It generally prefers capturing by shared reference. That is, by default, it tries to immutably borrow the variables from the outer scope.

Depending on the usage of the closed-over variables in the closure body, Rust may otherwise determine that mutable references should be taken instead, or that value semantics should be used, e.g., moving vs copying depending on their types. When the `move` prefix is explicitly used, those closures use value semantics, i.e., move or copy. These are explained in some more detail in the following few sections.

19.2. Capture Modes

If a closure expression, without the `move` prefix, refers to a)) from the environment, Rust follows the following rules to capture the variable, in the given order, starting from the top. If it can successfully capture the variable that allows the closure to compile with any of the rules (without regards to any other code outside the closure), then that rule is used for that specific variable:

- First, by immutable borrow,
- Then, by unique immutable borrow (see the next section),
- Next, by mutable borrow, and finally
- By value, e.g., copy or move.

If the `move` keyword is explicitly used, however, then *all* captures use value semantics:

- By move for the variables of Move types, and
- By copy for those of `Copy` types.

We will go over the capturing mode a bit more in the next couple of sections.

19.2.1. Unique immutable borrows

In Rust, the general rule regarding ownership and borrowing is that a value can be borrowed by one mutable reference or by one or more immutable references. When capturing an outside variable in a closure, however, there might be situations where the closure needs to capture the variable immutably but it cannot share it while the variable is captured. This is called the unique immutable borrow, and it is only used in this context.

A unique immutable borrow occurs in a capture when modifying a referent of a mutable reference. For example,

```
fn unique_immutable_borrows() {
  let mut mr = false;
  let im2mr = &mut mr;                    ①

  let mut c = || *im2mr = true;           ②
  // let can_i_borrow = &im2mr;           ③
  c();

  let _refref = &im2mr;                   ④
  assert_eq!(*im2mr, true);               ⑤
}
```

① The type of `im2mr` is `&mut bool`. But, note that the variable itself is *immutably* declared.

② We can modify the content that `im2mr` points to, and hence it is essentially a *mutable borrow* although syntactically it is not.

③ Therefore, we should not allow anybody else to take a (shared) reference of `im2mr`, that is, until the closure is done. That's what Rust does using the unique immutable borrow semantics. This line will cause a compile time error.

④ Now, we have no problem immutably borrowing `im2mr`.

⑤ Effectively, we ended up modifying the content of the immutable variable.

19.3. Move Closures

If you are familiar with lambda functions in other programming languages, then Rust's closures may seem overly complicated. In all other languages, we do not have to worry about "capture mode", for instance. This is because of Rust's ownership and borrowing model. Rust's closures generally work differently than lambda functions in other languages.

Rust's move closure is, however, what more closely corresponds to the lambda function. In general, a lambda function "captures" the closed-over variables, as in "really taking over the ownership". For this, captured variables are moved over e.g., from the stack, to a "safer" location, e.g., on the heap, so that they do not get deleted when their parent, e.g., a function, goes out of scope. Rust closures generally do not do that. As illustrated in the capture mode logic earlier, these variables are merely borrowed, if they can be.

Rust's move closure, however, really "captures" the external variables. As stated, depending on the type of the variable, it may be *move* or it may be *copy*. But, interestingly, in this particular context, that matters little. The external variable will likely go out of scope at some point. But, the closure already took over the variable, either through move or copy, and how it ended up capturing that

variable is not significant. The closure now owns the (possibly-copied) variable, and it can use it during its entire lifetime, regardless of the lifetime of the original scope the variable was originally captured in. Here's a simple example using move closures:

```
fn fib() -> impl FnMut() -> i32 {          ①
  let (mut a, mut b) = (0, 1);             ②
  move || {                                ③
    (a, b) = (b, a + b);
    b
  }
}
```

① This syntax is discussed later.

② `a` and `b` are local variables, e.g., local to the `fib` function.

③ This closure has to be a `move`. Why? (Note that the `fib` function returns this closure with the captured variables in it.)

```
fn move_closure_demo() {
  let mut f = fib();                       ①
  for _ in 1..10 {
    println!("{}", f());                   ②
  }
}
```

① The type of `f` is `impl FnMut() -> i32`. The impl trait type is discussed later. Note that we called `fib()` and it returned. All its local variables would have gone out of scope at this point. But, it matters not to the closure `f` since it "captured" the variables that it needed.

② What would be the output? ☺

19.4. Call Traits

Closure types all implicitly implement the `std::ops::FnOnce<Args>` trait, which means that all closures can be called at least once, e.g., by consuming ownership of the closure. In addition, some closures may also implement the `std::ops::FnMut<Args>` trait, which can be called by mutable reference, or the `std::ops::Fn<Args>` trait, which can be called by shared reference.

19.4.1. The `std::ops::FnOnce<Args>` trait

The `FnOnce` trait is implemented automatically by closures that capture variables by value, and it represents a method type that takes a by-value receiver, `self`. Instances of a `FnOnce` type can be called at least once.

19.4.2. The `std::ops::FnMut<Args>` trait

The `FnMut` trait is implemented automatically by closures that capture variables by mutable reference, and it represents a method type that takes a mutable receiver, `&mut self`. `FnMut` can be used as a trait bound for function-like types that need to be called more than once. `FnOnce<Args>` is a supertrait of `FnMut<Args>`,

19.4.3. The `std::ops::Fn<Args>` trait

The `Fn` trait is implemented automatically by closures that might capture variables by immutable reference. It represents a method type that takes an immutable receiver, `&self`. `Fn` can be used as a trait bound for function-like types that need to be called more than once without mutating state. Both `FnMut` and `FnOnce` are supertraits of `Fn`.

Chapter 20. Type Aliases

A type alias can be declared for an existing type using the keyword `type`. For example,

```
fn simple_type_aliases() {
  type Vital = (f32, i16);              ①
  let v: Vital = (37.5, 65);            ②
  println!(
    "Body temp:\t{},
Heart rate:\t{}",
    v.0, v.1                            ③
  );
}
```

① This declaration introduces `Vital` as a synonym for a tuple type `(f32, i16)`. Note the syntax, which is somewhat reminiscent of variable declarations.

② We can use this type alias in place of a type or type name.

③ The variable `v` refers to a tuple of the type `(f32, i16)`.

Type aliases can be declared with generic type parameters and/or as synonyms for types with generic type arguments. For example,

```
fn generic_type_aliases() {
  type IOResult<T> = Result<T, &'static str>;  ①
  type I32Result = IOResult<i32>;              ②
  let r1: I32Result = Ok(42);                  ③
  let r2: I32Result = Err("File error");
  println!("r1 = {r1:?}; r2 = {r2:?}");
}
```

① A type alias `IOResult<T>` is declared for an enum type `Result<T, &'static str>`. Note that `Result`'s second parameter is hard-coded with `&'static str`.

② `I32Result` is an alias to `IOResult<T>` with `T = i32`.

③ The type aliases `I32Result` and `IOResult<T>` can be used in place of types.

Types aliases are also commonly used for function types. For example,

```
type BoolFn = fn(bool) -> bool;          ①
fn do_boolean(f: BoolFn, a: bool) -> bool {
  f(a)
}
fn function_type_aliases() {
  let f1 = |_| true;                     ②
  let f2 = |_| false;
  let f3 = |a: bool| -> bool { a };      ③
  let f4 = |a: bool| -> bool { !a };
  for f in [f1, f2, f3, f4] {            ④
    for b in [true, false] {
      let x = do_boolean(f, b);
      print!("{x}\t");
    }
    println!();
  }
}
```

① A type alias for a function type, `fn(bool) -> bool`.

② Non-capturing closures can be coerced into function types.

③ This closure, for instance, is effectively the same as the function, `fn f3(a: bool) -> bool { a }`.

④ Note that both functions and closures are Copy types.

Chapter 21. The Struct Types

In Rust, the user can define a custom type using `struct`, `union`, or `enum`.

Struct A struct is a data structure consisting of a sequence of named or unnamed fields.

Union A (C-style) union is a type that can be interpreted as one of a few different types at run time.

Enum An enum is a union of one or more variants, each representing a disjoint set of values.

Values of Rust unions are only accessible in unsafe code, and hence we do not include unions in this book. We will go over basic syntax and semantics of the Rust struct in this chapter. Enums are discussed in the next chapter.

21.1. Structs

A `struct` in Rust is a multiplicative type similar to tuples. It represents a product of other types, called the *fields* of the struct type. Structs can be further divided into three groups, (regular) structs, tuple structs, and unit-like structs. Tuple structs and unit structs are described later in the chapter. A new struct type, that is, a regular struct with fields, is declared with the following general syntax:

```
struct IDENTIFIER {
  FIELDNAME1 : TYPE1,                    ①
  ...
  FIELDNAMEn : TYPEn,
}
```

① Fields are separated by commas. The comma after the last field is optional.

Structs can be declared generically as well. In such a case, generic parameters enclosed in `<>`, with an optional `where` clause for any trait bounds, are placed after the struct name `IDENTIFIER`. For example,

```
struct CarnivoreMeal<Meat: ?Sized> {      ①
  drink: u8,                              ②
  main_dish: Meat,
}
```

① Using the shorthand notation for the trait bound.

② A field, comprising a name and its type, separated by a colon (`:`). The order of the fields in struct declaration is not significant unless the `repr` attribute is used to fix the memory layout.

A `struct` can be public or non-public, just like all other items in a module. Likewise, each field of a struct can be public or non-public, e.g., based on the presence or absence of the `pub` visibility modifier. Public fields of a public struct can be accessed from outside the module. For example,

```
mod struct_demo {
  pub struct StructWithFields {           ①
    pub field_one: u8,                    ②
    field_two: i16,                       ③
  }
}
```

① A public struct, accessible from outside the `struct_demo` module. This type can be viewed as a product of two types, `u8` and `i16`.

② A public field, accessible from outside the module.

③ A private field. Note that a public struct with at least one private field cannot be constructed outside the module using the struct literal syntax, as we discuss in the next section.

Unlike some similar structures in other programming languages, the order of the fields in a Rust struct in terms of the memory layout may not be the same as the order of the fields, e.g., as defined in the declaration. This is, for instance, to allow for compiler optimization. The memory layout can be fixed using the `repr` attribute, but we do not discuss it in this book.

Struct Expressions

New instances of a struct, as well as instances of a struct variant of an enum, can be constructed using a struct literal syntax. (Technically, struct expressions are not really literals. In Rust, literals are single lexical tokens, like numbers or string literals. Nonetheless, it is not uncommon to call the struct construction expression a struct literal.)

A regular struct, or field struct, expression consists of the name or path to a struct, followed by a list of field name and value pairs, enclosed in the curly braces ({}). Note that, in general, all fields declared in the struct item need to be included in the struct literals. The fields can be specified in any order. For example,

```
mod demo {
  #[derive(Debug, Clone, Copy)]
  pub struct Drink {
    size: u8, // In ounces or in liters? ①
    kind: &'static str,
  }
  impl Drink {                          ②
    pub fn new(size: u8,
      kind: &'static str) -> Drink {    ③
```

```
      Drink { size, kind }                ④
    }
  }
}
fn new_drink_demo() {
  let d = demo::Drink::new(16, "diet water");
  println!("{d:?}");                        ⑤
}
```

① Note that the `Drink`'s fields are private in this example.

② An *inherent implementation* of `Drink`.

③ It's conventional to use associated `new` or `new_xxx` functions, or `from` or `from_xxx` functions, to create instances of structs.

④ This struct expression is equivalent to `Drink {size: size, kind: kind}`. When a value expression of a field is a variable with the same name as the field, e.g., `fieldname: fieldname`, that field can use the shorthand notation, e.g., `fieldname`.

⑤ This will print *Drink { size: 16, kind: "diet water" }*.

```
#[derive(Debug)]
struct ComboMeal<Meat, const N: usize> { ①
  drink: demo::Drink,                     ②
  burgers: [Meat; N],                     ③
}
fn order_super_meal() {
  let meal = ComboMeal {                  ④
    burgers: ["turkey", "goat", "tuna"], ⑤
    drink: demo::Drink::new(64, "beer"), ⑥
  };
  println!("{meal:?}");                   ⑦
}
```

① The `const` parameter is explained in an earlier chapter.

② Although `Drink` has all private fields, the type itself is public.

③ An example of using the `const` generic parameter.

④ A struct literal expression, e.g., to create an instance of `ComboMeal`. The type of `meal` is `ComboMeal<&str, 3>`.

⑤ Note that, as indicated, the order of the fields in the struct literal syntax is not important, regardless of whether the struct is declared with the `repr` attribute or not.

⑥ If the `Drink`'s fields were public, this `new` associated function call would have been equivalent to the struct expression, `drink: demo::Drink { size: 64, kind: "beer" }`. Note that we use the qualified path `demo::Drink`, and not just the name, in this case.

⑦ The output will be *ComboMeal { drink: Drink { size: 64, kind: "beer" }, burgers: ["turkey", "goat", "tuna"] }*.

21.2. Struct Update Syntax

A new instance of a `struct` type can be created based on an existing instance of the same type. This is sometimes called the *functional update syntax* (although technically we are constructing a new instance, and not updating an existing one).

This is another example of the constructs that originally started in functional programming languages and are becoming increasingly more popular in imperative languages.

Generally speaking, Rust's `struct` is comparable to the `data` type in Haskell, for instance, and especially to its `record` type syntax. More "mainstream" languages like C#, Java, and JavaScript/TypeScript have been likewise supporting various forms of record-like data types, as well as this *immutable update*, or "spread", syntax.

Rust uses more or less the same struct expression syntax, but with the `..` token (two dots) followed by another struct (known as the *base*) in the trailing field position, in the curly braces (`{}`). For example,

```
#[derive(Debug)]
struct Rec<'a> {
  f1: i32,                                  ①
  f2: String,                               ②
  f3: &'a str,                              ③
}
```

① Note that the primitive integer type `i32` is a `Copy` type.

② `String` is a Move type.

③ `&str` is intrinsically a reference type.

```
fn struct_update_demo() {
  let rec1 = Rec {                          ①
    f1: 100,
    f2: String::from("Warm"),
    f3: "Weather",
  };
  println!("{rec1:?}");                     ②
  let rec2 = Rec {                          ③
    f1: 105,
    f2: String::from("Hot"),
    ..rec1                                  ④
  };
  println!("{rec2:?}");                     ⑤
  let rec3 = Rec {                          ⑥
    f3: "Climate",
    ..rec1
  };
```

```
    println!("{rec3:?}");                    ⑦
    // println!("{rec1:?}");                 ⑧
}
```

① A struct literal expression to create a new instance of the `Rec` type, as defined above.

② This will print out *Rec { f1: 100, f2: "Warm", f3: "Weather" }*.

③ We are using the *struct update* expression syntax, using `rec1` as the *base*. The values of `f1` and `f2` are explicitly set, whereas the rest (only `f3` in this example) are set based on the corresponding values in `rec1`.

④ The syntax for specifying the base, with the `..` prefix. Note that the trailing comma is not allowed in this form.

⑤ An expected output: *Rec { f1: 105, f2: "Hot", f3: "Weather" }*

⑥ Another example of the functional update syntax. In this case, only `f3` is explicitly set or overwritten. The values of `f1` and `f2` are based on `rec1`.

⑦ An expected output: *Rec { f1: 100, f2: "Warm", f3: "Climate" }*

⑧ This will cause a compile error, *borrow of partially moved value*.

As can be seen from this simple example, Rust adds some twist to this now-widely used record construction syntax, e.g., due to the fact that Rust more commonly uses move-based value semantics.

As indicated, `i32`, the type of `f1`, is a `Copy`. On the other hand, `String` is a Move. In the example of the `rec3` struct literal, the value of `f1` is copied from `rec1`, whereas the value of `f3`, a `String`, is moved from `rec1.f3` to `rec3.f3`. Therefore, if we try to use `rec1` after this struct expression, we will run into an error. One thing to note is that, in order to be able to use this *immutable update syntax* outside the module, the fields of the given struct type must be all public. This is the same requirement as with the struct literal syntax.

21.3. Field Access Expressions

A field of a struct can be accessed using the dot operator (`.`) with the struct, followed by the field name. A field expression is a place expression that points to the location of the given field (e.g., like a variable). The field expression is mutable/immutable depending on whether the struct variable itself is mutable or not. For example,

```
#[derive(Debug)]
struct Foo {                                ①
  fighter: &'static str,
  number: fn() -> i32,                      ②
}
```

① An example struct with two fields.

② Note that the `number` field is callable.

```
fn foofoo() -> Foo {                        ①
  Foo {
    fighter: "Foo Fighters",
    number: || 21,
  }
}
```

① The `foofoo` function returns an instance of the `Foo` type.

```
fn field_access_demo() {
  let ff1 = foofoo().fighter;               ①
  println!("{ff1}");

  let ff2 = (Foo {
```

```
    fighter: "Bar Fighters",
    number: || 42,
  })
  .fighter;                                  ②
  println!("{ff2}");

  let mut foo = Foo {                        ③
    fighter: "Kung Foo",
    number: || 84,
  };
  foo.fighter = "Fung Foo Panda";            ④
  println!("{foo:?}");
  let num = (foo.number)();                  ⑤
  println!("{num}");
}
```

① Since `foofoo()` returns a struct, e.g., an instance of `Foo`, we can use the field access syntax.

② Another example of a field access expression. Note the use of the parentheses around the struct literal.

③ The variable `foo` is declared to be mutable.

④ Hence, its field access expression is also mutable.

⑤ Since the field `number` is callable, the field expression `foo.number` is callable. Note again the use of the parentheses. Without the parentheses, it would have been parsed as a method call expression.

21.4. Tuple Structs

A tuple struct can be viewed either as a tuple with named fields or as a struct with anonymous fields. In fact, tuple structs lie somewhere between tuples and structs in Rust, both syntactically and semantically.

A tuple struct type is declared with a struct name followed by a tuple type. For example,

```
struct NamedTuple(u16, i32, f64);
struct AnonymousStruct<'a>(bool, &'a str);
```

Note that, unlike tuples, the names of structs (including tuple structs) are part of the type definitions. That is, for example, `NamedTuple` above and the following tuple struct are two different types, although they consist of the same fields.

```
struct TuplelikeStruct(u16, i32, f64);
```

Tuple structs can be generically declared as well, e.g., as shown above with `AnonymousStruct<'a>`. Generic type and constant parameters can also be used. An instance of a struct tuple can be constructed using the tuple struct literal syntax. For example,

```
#[derive(Debug)]
struct Triple(u8, u8, i128);

#[derive(Debug)]
struct Pair<T1: Clone, T2: ?Sized>(T1, T2);

fn tuple_struct_demo() {
  let ts1 = Triple(0, 10, 100_000_000);  ①
  println!("{ts1:?}");

  let ts2 = Pair("Hi", 007);             ②
  println!("{ts2:?}");
}
```

① The syntax of the tuple struct expression is the struct name followed by its members, or anonymous fields, in a pair of parentheses. Note that their order is important since they are all unnamed.

② The inferred type of `ts2` is `Pair<&str, i32>`. One can also explicitly specify the type in the `let` binding, or by using the *turbofish syntax* with the tuple struct literal, e.g., `Pair::<&str, u8>("Hi", 007)`.

Alternatively, tuple structs can use the regular field struct expression syntax, using the indices, `0`, `1`, ..., instead of field names. We can also use the immutable update syntax. For instance, using the `Triple` example from above,

```
fn tuple_struct_demo_2() {
  let triple = Triple {
    0: 1,                                   ①
    1: 10,
    2: 1_000_000_000_000_000_000_000_000,   ②
  };
  println!("{triple:?}");

  let triple_again = Triple {               ③
    1: 255,
    .. triple
  };
  println!("{triple_again:?}");
}
```

① The order of the fields are not significant in this syntax.

② Incidentally, not many programming languages support 128-bit integer types. What is the biggest number you can *read*? ☺

③ Constructing a new instance of `Triple` based on an existing value `triple`.

21.5. The New Type Pattern

Most programming languages support a method to create a new type or type alias based on an existing type. Rust supports creating a type alias or synonym using the `type` keyword, with some limitations. For instance, as another example,

```
#[derive(Debug)]
struct Stock<T: Copy>(&'static str, T);
type Alias = Stock<f32>;                    ①

fn type_alias_demo() {
  let stock: Alias = Stock("Rust", 1.0); ②
  println!("{stock:?}");

  // let stock2 = Alias("Iron", 0.0);     ③
  // println!("{stock2:?}");
}
```

① `Alias` is a type alias to a tuple struct type `Stock<f32>`.

② The type alias `Alias` can be used just like a "real type". The type of `stock` in this declaration is `Stock<f32>`. The inferred type would have been `Stock<f64>`.

③ Rust's type alias has some limitations, however. For example, a type alias to a tuple struct type cannot be used as a constructor. This particular tuple struct literal expression, in this example, is not allowed in Rust.

In addition, some programming languages support creating a brand-new type, based on an existing type, which is in many ways similar, or almost identical, to the existing type and which is yet a different and distinct type from the original type. Go, for example, supports this using the `type` definition. Haskell likewise supports it through the `newtype` construct.

Rust does not have a builtin support for creating a new type this way, but one can use the tuple struct type for this purpose. In particular, one can create a tuple struct with one single field of a target type. This is often known as the "new type pattern" in the Rust community. For example,

```
#[derive(Debug, PartialEq)]
struct Identity(u64);                    ①
```

① `Identity` is a "new type" based on, but distinct from, the type `u64`. The idea here is, although we just need, or we could just use, `u64`, we define a separate type, e.g., for added benefits.

```
impl Identity {                          ①
  fn is_valid(&self) -> bool {
    if self.0 >= 1_000_000 && self.0 < 10_000_000 {
      true
    } else {
      false
    }
  }
}

use std::fmt::{Display, Formatter, Result};
impl Display for Identity {              ②
  fn fmt(&self, f: &mut Formatter<'_>) -> Result {
    write!(f, "{} ({})", self.0, self.is_valid())
  }
}
```

① We can add associated functions and methods to the type. In this particular example, since `u64` is a builtin type, we could not have added an extra method on `u64`. (And, this is one of the benefits of using "new types".)

② Likewise, we can implement a trait on this new type. In general, due to Rust's " orphan rule", it is not always possible to implement (possibly, somebody else's) traits on (likewise, possibly somebody else's) types. The "new type" idiom gives you more control in this regards.

```
fn can_i_pass(id: &Identity) {          ①
  if id.is_valid() {
    println!("You can pass");
  } else {
    println!("You shall not pass!");
  }
}
```

① Note that the new type pattern provides more type safety. In this (rather trivial) example, this `can_i_pass` function takes an argument of `Identity`, but not just a number of `u64` (although they are essentially the same in the current context).

```
fn newtype_pattern_demo() {
  let id1 = Identity(1234567);
  let id2 = Identity(1);
  println!("{id1}, {id2}");
  assert_ne!(id1, id2);                 ①

  for id in &[id1, id2] {
    can_i_pass(id);                     ②
  }
  // can_i_pass(123456789);             ③
}
```

① Since we derived `Debug` and `PartialEq` on `Identity`, we can compare the values of `Identity` using the `assert_eq` macro.

② The `can_i_pass` function can be called with a value of the `Identity` type.

③ This call, for example, will not compile.

Clearly, there is a tradeoff, and if the benefits outweigh the overhead, use of the new type pattern may be preferred over just using simpler types. One last thing. New type tuple structs, as well as other field- and tuple- structs in general, can use the destructuring syntax to obtain their field values, as we describe in a number of different places in the book. In case of new type structs with one base type, here's an example:

```
fn newtype_pattern_demo_2() {
  for id in &[Identity(7654321), Identity(1000000)] {
    if let Identity(1000000) = id {      ①
      println!("You are wanted! RUN!!!");
    } else {
      let Identity(u1) = id;             ②
      println!("Your internal ID = {u1}");
    }
  }
}
```

① The `if let` expression using the tuple struct pattern.

② The `let` binding also uses (irrefutable) patterns on the left-hand side.

21.6. Unit-Like Structs

A unit-like struct type is a special kind of struct type which includes no fields. Unlike field structs or tuple structs, a unit-like struct is declared with a name only, e.g., without any trailing curly braces or parentheses. When a unit-like struct type is declared, Rust automatically defines a constant of the given type with the same name. That is, the name of the type is also the name of the implicitly defined

constant, which is the only value of that unit-like struct type. For example,

```
#[derive(Debug, PartialEq)]
struct UnitCircle;                          ①

fn unit_struct_demo_1() {
  println!("{UnitCircle:?}");               ②

  let circle1 = UnitCircle;                 ③
  let circle2 = UnitCircle {};              ④
  assert_eq!(circle1, circle2);
}
```

① A unit-like struct type named `UnitCircle`.

② `UnitCircle` is also a constant value of the `UnitCircle` type.

③ The variables `circle1` and `circle2` have the same value.

④ Syntactically, `UnitCircle {}` is a struct literal, but it returns the same, and the only, value, `UnitCircle`.

Note that Rust's unit-like structs are comparable to singleton objects. Although unit-like structs do not have any fields, they can still be associated with functions or methods. For instance,

```
impl UnitCircle {
  fn radius() -> f32 {                      ①
    1.0
  }
  fn diameter(&self) -> f32 {               ②
    2. * UnitCircle::radius()
  }
}
```

```
fn unit_struct_demo_2() {
  let r = UnitCircle::radius();          ③
  println!("radius = {r}");
  let d = UnitCircle.diameter();         ④
  println!("diameter = {d}");
}
```

① An associated function.

② An associated method. Note that there is no real difference between functions and methods in case of unit-like structs.

③ Calling an associated function on the constant value.

④ Calling an associated method.

Builtin Attributes

The `deprecated` Attribute

Another rather commonly used attribute is `deprecated`. Although breaking backward compatibility should really be considered a *sin* ☺, in reality, it is a rather common practice. One can use the `deprecated` attribute to warn users (e.g., other programmers) for upcoming *radical*, and possibly backward-incompatible, changes. For example,

```
#[deprecated]
struct JustBornAndSad {
  bones: i32,
  brains: bool,
}
```

Chapter 22. The Enum Types

Enums, or enumerated types, in Rust are algebraic sum types, often called the disjoint unions, discriminated unions, tagged unions, or simply unions. (Not to be confused with Rust's C-style `union` type.) An `enum` consists of a number of *variants*, with each variant representing a (mutually exclusive) set of values. The overall `enum` type refers to a set union of all sets of values corresponding to these enum variants. (And, hence the name, the union or sum type.)

An `enum` item declares an `enum` type and *all* of its variants. Each variant has a name, and although variants themselves are not types, they use the syntax of the unit struct, tuple struct, or (regular) struct types.

22.1. Enum Variants

Enums are declared with the keyword `enum`, followed by the enum name and a list of its variant members, enclosed in curly braces (`{...}`). Enums can be generic. Each variant is declared using a struct-like syntax, which represents a constructor for that variant (e.g., using the struct literal syntax). Variants are also used in pattern matching for instances of the given enum. Each variant is associated with an integral discriminant value, and in case of unit-like variants, discriminant values can be explicitly given in an enum declaration.

For instance, here's a general syntax in an informal notation:

```
enum IDENTIFIER {
  VARIANT1,
  ...
  VARIANTn,
}
```

All variant names have to be unique within a given `enum`. Here are some concrete examples:

```
#[derive(Debug)]
enum PetA {                                    ①
  Puppy,
  Kitty,
}
fn unit_variants() {
  let pet1 = PetA::Puppy;                      ②
  let pet2 = PetA::Kitty;
  println!("My pets = {pet1:?} and {pet2:?}");
}
```

① An enum with two unit-like struct variants, `Puppy` and `Kitty`.

② An enum variant is essentially a constructor. Both `pet1` and `pet2` are variables of the type `PetA`.

```
enum PetB {                                    ①
  Puppy(&'static str),
  Kitty(u8),
}
fn tuple_variants() {
  let _p1 = PetB::Puppy("Snoopy");             ②
  let _p2 = PetB::Kitty(9u8);
}
```

① An enum with two tuple-like struct variants, `Puppy(&'static str)` and `Kitty(u8)`. Both happen to have one unnamed field.

② Enum variants are used to construct values of that enum. Both `_p1` and `_p2` are variables of the type `PetB`.

```
enum PetC {                                  ①
  Puppy { name: &'static str },
  Kitty { lives: u8 },
}
fn struct_variants() {
  use PetC::*                                ②
  let _p1 = Puppy { name: "Doo" };           ③
  let _p2 = Kitty { lives: 99 };
}
```

① An enum with two field struct variants, `Puppy{name: &'static str}` and `Kitty{lives: u8}`, each of which has one named field.

② The `use` statement can be sometimes useful to reduce clutter.

③ In all three cases, struct variants of an enum use the same corresponding struct expression syntax to construct instances of that enum.

Classification of variants into these three kinds are somewhat artificial, and these different kinds of variants can certainly be mixed in a single `enum` declaration. Note, however, that variants, `Cat` (a unit-like struct), `Cat()` (an empty tuple), and `Cat{}` (a struct with no fields), are all syntactically different from each other. Enum support iterations, and pattern matching over their variants, and so forth. Enums can be associated with constants, functions, and methods through trait declarations and/or implementations. Since these are common across different types in Rust, we don't specifically discuss these use cases for `enum` in this chapter.

22.2. Enum Discriminants

Each variant of an enum is associated with a discriminant value, starting from `0` and incrementing by `1` through all variants of the enum in the listed order. Their types are `isize` by default (although the compiler may choose to use a different/smaller type in the actual memory layout).

Depending on the use cases, these discriminants can be given custom values, and they can even be given different integer types. This customization can be achieved using an assignment-like syntax after a variant. For example,

```
enum Musketeer {
  Athos,                                  ①
  Aramis = 42,                            ②
  Porthos,                                ③
}
fn enum_discriminants() {
  let m1 = Musketeer::Athos;
  let m2 = Musketeer::Aramis;
  let m3 = Musketeer::Porthos;
  assert_eq!(0, m1 as u8);                ④
  assert_eq!(42, m2 as i16);
  assert_eq!(43, m3 as u32);
}
```

① The discriminant value of the first variant, `Athos`, is `0`, by default.

② We can explicitly set the variant's discriminant value, using this assignment syntax. In this example, the discriminant value of `Aramis` is `42`.

③ Then, without an explicitly assigned custom value, the discriminant value of `Porthos` is automatically set to an integer value that is `1` bigger than the previous value, which is `43` in this case. Note all discriminant values have to end up unique through this manual and automatic assignment.

④ We can get the discriminant values by explicitly casting the enum values to any compatible integer type. Note that the discriminant values of Rust's enums are always integers, unlike in some other languages which support strings and what not. On the other hand, Rust's enums are much more general (comparable to `data` types in Haskell, for instance), and their variants can potentially contain other data values, including strings, etc.

In enums like this, which consist of only unit-like/fieldless variants, assigning, and accessing, custom discriminant values is straightforward. For enums with compound variants, e.g., tuple-like or struct-like variants, the `enum` has to be explicitly annotated with the `repr(INTTYPE)` attribute. to be able to use custom discriminant values or use specific integer types. For example,

```
enum Desert {
  // Apple = 5,                          ①
  Coffee(bool), // Decaf?
  IceCream { flavor: String },
}
```

① This will cause an error although this variant itself does not contain any data.

Furthermore, in case of general union-type enums, their variants cannot be simply cast or coerced to integer values in safe Rust.

```
#[derive(Debug)]
#[repr(u16)]                              ①
enum Humanlike {
  Person = 100,     // Average IQ
  Chatbot { maker: String } = 20,         ②
  // Terminator(f64) = 65550,             ③
}
fn enum_discriminants_2() {
  let h1 = Humanlike::Person;
  let h2 = Humanlike::Chatbot {
    maker: String::from("ClosedAI"),
  };
  // println!("h1 type: {}", h1 as u16); ④
  println!("h2 = {h2:?}");
}
```

① This attribute, with a specific integer type, is needed to be able to use custom discriminant values.

② The custom discriminant values can be given to a compound variant as well.

③ Incidentally, since we declare this enum to use `u16`, `65550` is not a valid discriminant value.

④ The general enum's discriminant values cannot be accessed in the safe Rust, e.g., by simply casting it to an integer type.

As this example illustrates, enums consisting of only fieldless variants have special uses in Rust, similar to those found in other C-style languages.

Builtin Attributes

The `repr` Attribute

The `repr` attribute is primarily used to control the low-level memory layout of composite types such as structs, enums, and unions. First of all, if you are coming from high-level languages, it's important to note that the memory layout is always *linear*. For example, a particular struct may appear to be organized in a tree, e.g., since a field of a struct may contain other fields, etc. But, ultimately, they are flattened (e.g., through a depth-first traversal) and they occupy a linear contiguous space on stack. (Some fields may point to memory in other locations, however.)

Most computers use 32 bit or 64 bit architecture, and the overall size of a value and its memory layout, e.g., especially across word boundaries, can affect the performance. This is where `repr` can be used. Otherwise, most Rust programmers working on high-level applications need not, and should not, generally worry about these details.

22.3. Zero-Variant Enums

There are a special kind of enums that have no variants. They are called the *zero-variant enums*, and they cannot be instantiated since they do not have variants. Zero-variant enums are often used as placeholder types, and they sometimes have overlapping use cases with the never `!` type.

```
enum Zero {}                                ①
fn zero_variant_enums_1() -> Zero {         ②
  loop {
    println!("It's not illegal!");
  }
}
fn zero_variant_enums() {
  let _z: Zero = panic!("Don't panic!"); ③
}
```

① A zero-variant enum, with a *very imaginative* name, `Zero`. ☺

② The `loop` expression never returns.

③ Likewise, the `panic!` macro does not return a valid value.

Clearly, there can be practically an infinite number of zero-variant enums (with different names), and they are all different types from each other.

```
enum Cero {}                                ①
fn zero_variant_enums_2() {
  let x: Zero = panic!("Panic!");
  // let y: Cero = x;                       ②
}
```

① Another zero-variant enum type, with a different name.

② This will cause a compile error due to mismatched types, besides the fact that this line is not reachable since `panic!` diverges (e.g., it returns with `!`).

Unlike `!`, zero-variant enums are "real" types. Zero-variant enums are, in a sense, comparable to the static classes in C#, for instance. In particular, although we don't see many such uses in the wild, Rust's zero-variant enums can be used to simply "organize" functions into groups, like the way static classes and empty structs are used in C#, Java, and Go, etc. For example,

```
enum Ghost {}                                ①
impl Ghost {                                 ②
  fn boo() { println!("I boo!") }
  fn doo() { println!("I doo!") }
}
fn another_zero_variant_enum_demo() {
  Ghost::boo();
  Ghost::doo();
}
```

① Another different zero-variant enum type.

② A zero-variant enum can be used to collect related (associated) functions into a namespace, if you will.

Builtin Attributes

The `non_exhaustive` Attribute

A struct initially defined with a certain set of fields, or an enum with a certain set of variants, may end up with more fields/variants in the future. This can cause backward compatibility issues if these types are publicly exported.

To help alleviate this potential issue, one can mark their types with the `non_exhaustive` attribute. In particular, structs, enums, and enum variants can be annotated as `non_exhaustive`.

Within the same crate where these types are defined, the `non_exhaustive` attribute has no effect. But, their uses will be limited in the outside crates. For example, a `non_exhaustive` struct may not be created using a struct expression. Likewise, enumerating over *all* variants of a `non_exhaustive` enum does not count as *exhaustive*. This is in order to preserve the backward compatibility in the future.

For instance,

```
#[non_exhaustive]
pub struct Rainbow {
  pub red: u32,
  pub blue: u32,
  // That's it for now, but
  // I may remember some more rainbow colors later...
}
```

```
pub enum Bug {
  #[non_exhaustive]
  Ant { head: bool, legs: u8, },
  #[non_exhaustive]
  Bee(bool),
  #[non_exhaustive]
  Scorpion,
}
```

Chapter 23. Smart Pointers

It's worth repeating that all values in Rust are stack allocated by default. In fact, there is no way to explicitly store values on the heap using the safe Rust language constructs only (e.g., like the `new` and `delete` operators in C++). (In contrast, in many higher-level (and, garbage-collected) languages like Python, JavaScript, and Java, values are all heap-allocated, except primitive type values like numbers.)

This is because the heap memory is often the cause of memory leak, and *all other bad things*. In fact, *heap memory is the root of all evil!!!* As long as we don't use the heap, we should be fine. ☺ Well, obviously, that's like throwing the baby out with the bath water. It's really not feasible to create any (non-trivial) software without using heap memory. Rust provides an *escape hatch*, if you will. Rust provides a number of types in the standard library, whose main purpose is to allow us to use the heap memory in a safe(r) way.

For example, as mentioned, the `String` and `Vec` types use heap memory. They are dynamic-sized, and they cannot be allocated on stack, and hence we need to rely on Rust to provide types like them.

Another commonly used type is `Box<T>`. Values can be explicitly stored on the heap using `Box<T>`. This is generally known as "boxing" in programming. Boxed values can be dereferenced using the star `*` operator in Rust, and the values themselves act like references or pointers. Types like `Box<T>` are often called *smart pointers* for this reason. They are "smart" because they manage their own memory, and the developers do not generally have to worry about allocating and de-allocating memory on the heap. (BTW, `Box<T>`, which corresponds to `unique_ptr` in C++, is one of the quintessential move-only types. That is, copy would not make sense for the values of `Box<T>` and other similar types.) We go over a few (smart) pointer types of Rust next. Note that the purpose of this chapter is to list a few important heap-based types so that the readers who are new to Rust know what to look for, and not to be comprehensive.

23.1. The `Box<T>` Struct

The generic `std::boxed::Box<T>` type is a container type, which is commonly used as a way to store values of an arbitrary type `T` on the heap. When a `Box` goes out of scope, its `drop` method is called, and the memory on the heap is freed.

```
pub struct Box<T: ?Size>;
```

The `Box<T>` struct has a few special features:

- The `*` operator for `Box<T>` produces a place which can be moved from,
- Methods can take `Box<Self>` as a receiver, and
- A trait may be implemented for `Box<T>` in the same crate as `T`, which is not possible, in general, for normal generic types, a la *orphan rule*.

23.1.1. Constructor

```
fn new(x: T) -> Box<T>;
```

The `new` constructor function allocates memory on the heap and then places the given value `x` into that location.

23.1.2. Examples

```
#[derive(Debug, Clone)]
struct Jack {
  clown: Option<Box<Jack>>,              ①
  age: u8,
}
```

```
impl Jack {
  fn new() -> Jack {
    Jack {
      clown: None,
      age: 1,
    }
  }
  fn push_one(&mut self) {
    self.clown = Some(Box::new(Jack {
      clown: self.clown.clone(),
      age: self.age,
    }));
    self.age = self.age + 1;
  }
}
```

① Types like `Jack` are called *recursive types*. Because their sizes cannot be known at compile time, they have to use pointer types like references, e.g., `&Jack`, or smart pointers, e.g., `Box<Jack>`.

The type `Jack` actually represents a singly linked list (e.g., with the `clown` field as the "next" pointer). One can easily implement the `push` and `pop` methods on `Jack`, e.g., something like `push(&mut self, jack: Jack)` and `pop(&mut self) -> Option<Jack>`. Another way is to use an `enum`, for instance,

```
#[derive(Debug, Clone)]
enum Node<T: Clone + Default> {
  Cons(T, Box(Node<T>)),                ①
  Nil,
}
```

① Again, this is a recursive type, and we need a pointer type here. The readers are encouraged to try and implement the `push` and `pop` methods, for practice.

23.2. The `Rc<T>` Struct

The `std::rc::Rc<T>` type is a "reference counting" smart pointer. It can only be used in a single-threaded environment. There is also a corresponding reference counting type in `std::sync`, called `Arc<T>`.

```
pub struct Rc<T: ?Sized> {
  /* private fields */
}
```

`Rc<T>` is generally used when multiple ownership is needed. Technically, `Rc` is *self-owned*, but conceptually, every reference added to `Rc` is considered an owner. A value of `Rc` keeps track of the number of references (or, "owners"), and when all references are removed, the `Rc` itself is destroyed. More specifically, the reference count of an `Rc` increases by `1` whenever the `Rc` is cloned, and decreases by `1` whenever one cloned `Rc` goes out of the scope. When the `Rc`'s reference count becomes zero, which means that there are no more owners remained, both the `Rc` value and its contained value are dropped.

23.2.1. Constructor and `Clone::clone`

```
fn new(value: T) -> Rc<T>;
```

```
fn clone(&self) -> Rc<T>;
```

As explained, cloning an `Rc` does not perform a deep copy. Cloning creates just another pointer to the wrapped value, and it increments the strong reference count. `Rc<T>` also implements the `Deref` trait (but, not `DerefMut`), and hence it can be dereferenced.

23.2.2. Examples

```
use std::rc::Rc;

#[derive(Debug)] struct Poly(u8);

fn count(arg: &Rc<Poly>) -> usize {
  Rc::strong_count(&arg)
}
pub fn rc_demo() {
  let rc1 = Rc::new(Poly(1));            ①
  let rc2 = rc1.clone();                 ②
  let rc3 = rc2.clone();
  println!("{c}", c = count(&rc3));      ③
}
```

① `Rc::new` creates a new `Rc` instance with the reference count `1`.

② Every cloning increases the reference count by `1`.

③ This will output *3*.

23.3. The `Cell<T>` Struct

A `Cell<T>` value represents a mutable memory location. This type, as well as `RefCell<T>`, are not technically smart pointers. They do not implement the `Deref` trait, and hence they cannot be *dereferenced*. Nonetheless, they have much in common with smart pointer types like `Box<T>`.

```
pub struct Cell<T: ?Sized> {
  /* private fields */
}
```

23.3.1. Constructor

```
fn new(value: T) -> Cell<T>;              ①
```

① The `new` function creates a new `Cell` containing the given `value` of type `T`.

23.3.2. Examples

```
pub fn cell_demo() {
  let celled = std::cell::Cell::new(10); ①
  println!("{celled:?}");
  celled.set(1000);                      ②
  println!("{celled:?}");
}
```

① A `Cell` is created using the `Cell::new` constructor with a value `10`. Note that we declare `celled` as `let`, and not `let mut`.

② But, we can still update the wrapped value of `celled`. This is called the "interior mutability".

23.4. The `RefCell<T>` Struct

The `RefCell<T>` type also represents a mutable memory location. It is similar to `Cell<T>`, but values of `RefCell<T>` act more like references, with dynamically checked borrow rules at run time.

```
pub struct RefCell<T: ?Sized> {
  /* private fields */
}
```

23.4.1. Constructor

```
fn new(value: T) -> RefCell<T>;          ①
```

① The `new` constructor function creates a new `RefCell` containing the given `value` of type `T`.

23.4.2. Examples

```
use std::cell::RefCell;
fn refcell_demo() {
  let mut c = RefCell::new(1);           ①
  *c.get_mut() += 1;                     ②
  assert_eq!(c, RefCell::new(2));
  let x = c.replace(10);                 ③
  assert_eq!(x, 2);
  assert_eq!(c, RefCell::new(10));
}
```

① A constructor call, with an interior value `1`, whose type is `i32`.

② The `get_mut` method returns a mutable reference of type `&mut T`, or `&mut i32` in this example, which can be used to update the value this reference points to. After this statement, the contained value of `c` will be `2`. Again, it demonstrates interior mutability.

③ The `replace` method replaces the interior value with the provided value, and it returns the old value. Hence, `x` will be `2`, and the RefCell `c` now will end up containing a new value `10`.

Note that the types listed in this chapter are all Move types, and they all implement the `Clone` trait.

Chapter 24. Traits

As we emphasize throughout this book, traits in Rust play two important roles, among other things. First, a trait divides all types into two mutually exclusive sets. A type either belongs or does not belong to the given trait class. Traits themselves are not types.

Second, a trait defines the common behavior for all types belonging to that trait class. An instance type of a trait needs to implement all functions and methods required by that trait (e.g., without default implementations), as well as any other required associated items such as constants and types.

24.1. Trait Declarations

A new trait can be declared as follows:

- The `trait` keyword,
- The name of the trait,
- Optional generic parameters, and
    - Trait bounds, including lifetime bounds, if any, and
- A block (`{}`), enclosing
    - Any inner attributes, and
    - Zero or more associated items.

The set of associated items, if any, defines the abstract interface of the trait that types need to implement to be part of that trait class. Associated items can be constants, type aliases, or functions/methods, with or without definitions (or, default implementations). They are further discussed in the next chapter. Impls, both inherent and trait implementations, are discussed in the following chapter.

For example,

```
trait MyTraitOne {
  const C1: i32 = 0b1000_000_000;          ①
  const C2: &'static str;                  ②
  type T1;                                 ③
  type T2<S>;                              ④
  fn f1() -> u8 { 255 }                    ⑤
  fn f2(a: i64, b: i64) -> Option<i64>;    ⑥
}
```

① A constant with a default value, associated with the type, `MyTraitOne`.

② An associated constant without a default value.

③ An associated type. Note that the associated type cannot have a default implementation, as of this writing.

④ An associated type can be generic even when the trait may not be generic.

⑤ An associated function with a default implementation.

⑥ An associated function without implementations.

A trait specified with generic parameters is a generic trait. The parameters appear after the trait name, using the same syntax used in generic types. For instance,

```
trait Drone<T: Copy> {
  type Wing<K: Copy>;
  fn new() -> Self;                        ①
  fn fly(&self, speed: f32) -> T;
  fn test(&self) -> Result<Self::Wing<T>, ()>;
}
```

① `Self` refers to the implementing type.

24.2. Supertraits

Rust doesn't support *type inheritance*. But, traits can be used to provide relationships among the types that implement those traits.

A trait can be declared with one or more *supertraits*. In order for a type to be able to implement a trait it needs to first implement all supertraits of the given trait, if any. Alternatively, the trait - supertrait relationship creates kind of subtype - supertype relationships among the implementing types. Syntactically, supertraits are declared by trait bounds on the `Self` type of a trait. Traits included in those trait bounds can also include supertraits of their own, and hence the supertrait specification is transitive.

For instance, here's a simple trait with one associated method, which includes a default implementation:

```
trait Decapod {
  fn legs(&self) -> u8 { 10 }
}
```

Now, we can create another trait as a subtrait of `Decapod` as follows:

```
trait Crustacean: Decapod {              ①
  fn walk_how(&self) -> &'static str;    ②
}
```

① This supertrait specification is a shorthand for `trait Crustacean where Self: Decapod { ... }`, as just described, and as explained earlier in the book, e.g., in the generics chapter.

② This crate defines a single associated method for illustration.

```
fn walk_with_legs<T: Crustacean>(d: T) {
  let how = d.walk_how();                ①
  let legs = d.legs();                   ②
  println!("I walk {how} with {legs} legs!");
}
```

① Since the generic parameter `T` is constrained to be `Crustacean`, we can call this method on a variable `d` of type `T`.

② Since `Decapod` is a supertrait of `Crustacean`, we can also call this method defined in `Decapod` on `d`. This is somewhat reminiscent of how type inheritance works in many OOP languages.

Here's a simple example of implementing a trait which has a supertrait.

```
struct Crab;
impl Decapod for Crab { }                ①
impl Crustacean for Crab {               ②
  fn walk_how(&self) -> &'static str { "sideways" }
}
```

① In order to be able to implement a trait, e.g., `Crustacean`, the type is *required* to implement all its supertraits as well, e.g., `Decapod` in this example. Note that, in Rust, "implementing" involves an explicit `impl` declaration, e.g., even when the trait may be empty.

② Implementing the `Crustacean` trait for `Crab`.

```
fn supertraits_demo() {
  let crab = Crab;
  walk_with_legs(crab);
}
```

Chapter 25. Associated Items

Certain kinds of items can be declared, and/or defined, in traits or implementations, and they are called the associated items. Currently, the following items can be associated with types:

- Constants,
- Types, and
- Functions and methods.

Each associated item kind comes in two varieties:

- Declarations that declare signatures, and
- Definitions that also contain the implementation of the declarations.

25.1. Associated Constants

Constants can be associated with a type, e.g., by declaring them in a trait which the type implements. The syntax is the same as the constant item declaration.

```
const IDENTIFIER : TYPE ;
const IDENTIFIER : TYPE = VALUE;
```

For example,

```
trait SSec {
  const NUMBER: i128;                    ①
  const IS_DUE: bool = false;            ②
}
```

① An associated `const` declaration, without an implementation.

② An associated `const` declaration with a default value.

```
struct SSecCard;                          ①
impl SSec for SSecCard {
  const NUMBER: i128 = 123_45_6789;       ②
  const IS_DUE: bool = true;              ③
}
```

① A unit-like struct, which is explained later.

② Associated constants without default values need to be implemented in an implementation by each implementing type.

③ Associated constants with default values can be overridden.

```
impl SSecCard {
  const VALID_UNTIL: u16 = 2025;          ①
}
```

① Associated constants can also be defined in an implementation. The constant `VALID_UNTIL` is now associated with the type `SSecCard` in this example. Note that constants declared in an `impl` need to have their values defined as well.

```
fn associated_constants() {
  println!("My social security number is {}",
    SSecCard::NUMBER);                    ①
  println!("Is due? {}",
    SSecCard::IS_DUE);                    ②
  println!("Valid until: {}",
    SSecCard::VALID_UNTIL);               ③
}
```

① We can refer to the associated constant using a path syntax.

② Ditto. Associated constants declared in a trait can also be referred to with their full paths. For example, `SSecCard::IS_DUE` is the same as `<SSecCard as SSec>::IS_DUE`. The full path syntax can be useful when constants with the same name exist in multiple traits that the type implements.

③ An associated constant declared and defined in an `impl` can also be referred to by its path. Note that the constant name alone, e.g., `VALID_UNTIL`, cannot be used to reference the associated constant.

25.2. Associated Types

Types, as a form of type aliases, can be associated with a type by declaring them in a trait which the type implements. Associated types cannot be declared in inherent implementations. They cannot be given default implementations with the declarations in a trait.

An associated type is declared within a trait, using the type alias declaration syntax. Unlike in the nominal type alias declarations, however, associated types can include an optional list of trait bounds, e.g., including the `where` clause.

For example,

```
trait Maxi {
  type Elem;                                ①
  fn maxi(&self) -> Option<Self::Elem>;     ②
}
```

① An associated type declaration, without an implementation, in this example.

② This method uses the associated type as a return value. The path syntax `Self::Elem` refers to the associated type `Elem` of the implementing type, which is referred to as `Self`.

```
impl Maxi for &[i32] {                          ①
  type Elem = i32;                              ②

  fn maxi(&self) -> Option<Self::Elem> {        ③
    self.iter().max().copied()
  }
}
```

① We can implement the `Maxi` trait for the `&[i32]` slice type, as an example. Note that we could have implemented it more broadly on a generic type like `&[T]`, e.g., with an appropriate trait bound on `T`.

② The type needs to provide an implementation for the associated type `Elem` since the trait does not include a default implementation.

③ The method needs to be implemented as well. Note that `Self::Elem`, used as a type parameter, is a shorthand for `<&[i32] as Maxi>::Elem`, which is `i32`.

```
impl Maxi for &str {                            ①
  type Elem = char;                             ②

  fn maxi(&self) -> Option<Self::Elem> {        ③
    self.chars().max()
  }
}
```

① Another trait implementation example, using `&str` this time.

② The associated type `Elem` is defined to be `char` in this case. Implementing an associated type uses the type alias syntax. That is, `Elem` is a type alias to `char` in the context of this implementation. Note that `&str` is not a generic type, at least syntactically, although conceptually it is sort of a collection type.

③ `Self::Elem`, or `<&str as Maxi>::Elem`, now refers to `char`.

Here's an example use of the `Maxi::maxi` methods.

```
fn associated_types() {
  let s1: &[i32] = &[21, 31, -11];
  let m1 = s1.maxi().unwrap();          ①
  println!("m1 = {m1}");
  let s2 = "HeLlO, cRaP!";
  let m2 = s2.maxi().unwrap();          ②
  println!("m2 = {m2}");
}
```

① The type of `m1` is `i32`, which is the associated type `Elem` of the type `s1`, `&[i32]`.

② The type of `m2` is `char` in this case.

Generic Traits vs Traits with Associated Types

What's the difference between traits with associated types and traits with generic parameters? The short answer is, there is no *real* difference. They are clearly different from each other from the syntactic point of view, but nonetheless both are constructs used to create generic, or *parameterized*, traits, that is, traits parametrized over other types.

Traits defined with an associated type are also inherently generic, and one can pretty much use one or the other when a type-parametrized trait is needed. There *are* differences, however. It is not something you can learn from a bullet point list, and the readers will need to pick this up through experience, but here are a couple of important points.

First, at the syntactic level, generic trait syntax can be used for generic type or constant value parameters, whereas the associated types are just types.

But, more importantly, a type that implements a generic trait that has a generic type parameter needs to be generic itself and, in general, it needs to use the same generic type parameters used in the trait, e.g., in addition to any additional generic type parameters. On the other hand, there is so such requirement for (syntactically non-generic) traits declared with associated types. The type only needs to "implement" this associated type(s), e.g., by providing a concrete type. This concrete type does not need to be a type that is related to the implementing type in any way (although they typically are in practice).

There are many other differences (conceptual or otherwise), but as stated, it is something each programmer will have to learn through experience. In many situations, the choice of one vs the other boils down to a personal preference.

If anything, many Rust programmers seem to find using traits with associated types more "natural" than using generic traits. For instance, refer to the example from the previous section, the `Maxi` trait. The concrete type for `Elem` is to be best determined by each implementing type rather than through a generic declaration on the trait. As we briefly discuss at the end, iterator-related traits defined in the standard library generally use associated types rather than the generic type parameter syntax, and it will be instructive to think about why that is the case.

25.3. Associated Functions

Functions associated with a type are generally referred to as "methods" of the type, e.g., as the term is commonly used in object oriented programming. A special class of functions associated with an instance of a type, called the *associated methods*, are described in the next section. All others are simply called the associated functions of the type.

The associated function declaration uses the similar syntax as the function item declaration, but associated functions are not allowed to be `async` or `const`. An associated function can be declared/defined in a trait and/or in an implementation for the type. When it is declared in a trait, its body can be, and often is, omitted.

For instance,

```
struct Gold {                                   ①
  karat: u8,
}
```

① An example type with one field.

A trait declaration with two associated functions:

```
trait Digger {
  fn new() -> Self;                             ①
  fn lucky() -> bool {                          ②
    true
  }
}
```

① A function declaration without an implementation. It uses the conventional name `new`, which is commonly used for functions that return a value of the type `Self`. As mentioned, `Self` refers to the implementing type, e.g., in an inherent or trait implementation.

② A function declaration with a default implementation.

```
impl Gold {
  fn new(karat: u8) -> Gold {                   ①
    Gold { karat }
  }
}
```

① An associated function with the type `Gold`. It uses the same conventional name `new`. The inherent implementation is discussed in the next chapter.

```
impl Digger for Gold {
  fn new() -> Self {                    ①
    Gold::new(24)
  }
  // fn lucky() -> bool { false }       ②
}
```

① The `Digger::new` function is now associated with the `Gold` type.

② The same with `Digger:lucky`. We can optionally override its default implementation.

```
fn associated_functions() {
  let gold1 = Gold::new(14);            ①
  let gold2 = <Gold as Digger>::new();  ②
  println!("Gold1 = {}k", gold1.karat);
  println!("Gold2 = {}k", gold2.karat);
  println!(
    "Are you feeling lucky? {}!",
    if Gold::lucky() {                  ③
      "Yes, very much"
    } else {
      "No, not at all"
    }
  );
}
```

① Calling an associated function of `Gold`.

② Because the same names are used, we will need to use the fully qualified name for `Digger::new` here.

③ The `Gold::lucky()` call is the same as `<Gold as Digger>::lucky()`.

25.4. Associated Methods

Associated functions which satisfy the following conditions can be invoked using the method call operator (`.`), for example, `x.foo(bar)`, instead of, or in addition to, the usual function call notation `x::foo(self, bar)`. They are called associated methods.

- The first function parameter, called its *receiver*, is `self` or `mut self`, or
- Its type, if explicitly specified, is one of the following:
  - `S`, `&'lt S`, `&'lt mut S`, `Box<S>`, `Rc<S>`, `Arc<S>`, or `Pin<P>`,
    - where `S` is either `Self` or the name of the implementing type, and
    - `'lt` represents a lifetime.

When the type is not explicitly specified, the following shorthand can be used:

- `fn f(self, …)` ← `fn f(self: Self, …)`.
- `fn f(mut self, …)` ← `fn f(mut self: Self, …)`.
- `fn f(&'lt self, …)` or `fn f(&self, …)` ← `fn f(self: &'lt Self, …)`.
- `fn f(&'lt mut self, …)` or `fn f(&mut self, …)` ← `fn f(self: &'lt mut Self, …)`.

Note that lifetimes are usually elided with this shorthand notation.

Here's a quick example:

```
struct Burrito<T: Copy + Default>(T);    ①
```

① A simple tuple struct for demo.

```
trait Wrap {
  type Core: Copy + Default;              ①
  fn unwrap(&self) -> Self::Core {        ②
    Self::Core::default()
  }
  fn rewrap(&mut self) -> Self;           ③
}
```

① Associated types can be specified with trait bounds.

② An associated method with a default implementation.

③ Another associated method without a default implementation.

```
impl<T: Copy + Default> Burrito<T> {      ①
  fn new(core: T) -> Self
  { Burrito::<T>(core) }                  ②
}
```

① The `Burrito` type implements a `new` constructor function.

② Note that `core` is a value of a Copy type.

```
impl<T: Copy + Default> Wrap for Burrito<T> {  ①
  type Core = T;                          ②
  fn unwrap(&self) -> Self::Core          ③
  { self.0 }                              ④
  fn rewrap(&mut self) -> Burrito<T> {    ⑤
    let rewrapped = Burrito::<T>::new(self.0);
    self.0 = T::default();                ⑥
    rewrapped
  }
}
```

① The generic type `Burrito` implements `Wrap`.

② `Burrito` includes an implementation for `Wrap::Core`, which is set to its generic parameter `T`. Note that this is a type alias syntax, and it cannot include trait bounds.

③ We overwrite the default implementation of `Wrap::unwrap`. The type `Self::Core` can be more explicitly written as `<Burrito<T> as Wrap>::Core` in this context.

④ Note that `self.0` will be copied when the `unwrap` method call returns.

⑤ Implementation of the associated method `rewrap` is required since it does not have a default implementation.

⑥ Just to illustrate a use of an associated method with `&mut self` parameter, we reset the old wrap's core to its default value.

```
fn associated_methods() {
  let mut b1 = Burrito::new(42);
  let u1 = b1.unwrap();                  ①
  assert_eq!(42, u1);
  let b2 = b1.rewrap();                  ②
  assert_eq!(42, b2.unwrap());
  assert_eq!(0, b1.unwrap());
}
```

① The associated method `unwrap` of `Burrito` can be used with the method call operator. This expression is equivalent to `Burrito::unwrap(&b1)`.

② The same with the `rewrap` associated method. This call expression is equivalent to `Burrito::rewrap(&mut b1)`.

Chapter 26. Implementations

An `impl`, short for implementation, is an item that declares associations of certain items with a given type and/or that provides implementations of those associated items. There are two kinds of implementations:

- An inherent implementation, for a type, and
- A trait implementation, for a pair of trait and type.

Impls are defined using the keyword `impl`, followed by a trait name and the keyword `for` in case of trait implementations, the name of the implementing type, and a block, which contains the definitions of the associated items such as constants and functions (e.g., functions or methods).

Functions are statically associated with a type, whereas methods are implemented for an instance of the type. Methods can be further classified into three kinds based on their receivers, e.g., `self`, `&self`, and `&mut self`.

26.1. Inherent Implementations

Items can be associated with, and implemented for, a type, e.g., independently of other traits. This is called the *inherent implementation* of the type. Inherent implementations can be provided only in the same crate that the type is declared in. That is, as a trivial corollary, one can only provide implementations for the custom types that they create.

An inherent implementation is defined with the `impl` keyword followed by a path to an implementing type, and a set of associated items enclosed in curly braces. The associated items can be constants or functions (including methods), but not type aliases. In case of generic impls, the generic parameters are placed after the `impl` keyword, and the `where` clause, if any, is included after the type.

Note that, syntactically, a type can have multiple inherent implementations, e.g., across different modules in the same crate.

26.2. Trait Implementations

A *trait implementation* is similar to an inherent implementation, but it can only include the definitions of the associated items declared and/or defined in the corresponding trait declaration, which is called the *implemented trait*.

A trait implementation must define all non-default associated items declared by the implemented trait, may redefine default associated items defined by the implemented trait, and cannot associate, or define, any other items with the implementing type.

Syntactically, a trait implementation is rather similar to an inherent implementation except that the implemented trait name followed by `for` is included between the `impl` keyword and the implementing type.

> To some people who are new to Rust, the syntax can be a bit confusing, or it may appear somewhat inconsistent between the *inherent* and *trait* implementations. (E.g., in one case, `impl` is followed by a trait whereas in the other, `impl` is followed by a type.) One way to reconcile this apparent "inconsistency" is to consider the inherent implementation syntax as a special case of the trait implementation. That is, `impl TYPE` can be viewed (conceptually) as `impl (ANONYMOUS-TRAIT for) TYPE`.

Another difference between the inherent implementation and trait implementation is that, since a `trait` can include associated types, a trait implementation can implement, or override, any associated type (or, type alias) declarations in the implemented trait.

Chapter 27. Dynamic Dispatch

As stated, Rust's traits are not types. In particular, types have memory implications, whereas traits do not.

27.1. Trait Objects

Traits can be used in certain places where types are expected. They are called the *trait objects*, and syntactically, they are denoted by a trait name preceded by the keyword `dyn`. Any number of optional auto traits can be specified after this base trait name, e.g., using the trait bound syntax. No more than one lifetime bound can be specified as well, if needed, using the same `+` operator. For example, the following are all syntactically valid trait objects, given a trait `MyTrait`:

- `dyn MyTrait`
- `dyn MyTrait + Send + Sync`
- `dyn MyTrait + 'static`
- `dyn MyTrait + Unpin + 'static`

A trait object is an (unspecified) type, or a value of such a type, that implements the specified *base trait* and its supertraits, as well as the auto traits, if any is specified.

The purpose of trait objects is to allow *late binding* of methods (e.g., at runtime). Normally, Rust only allows *early binding* of methods (e.g., at compile time). But, calling a method on a trait object results in *dynamic dispatch* at runtime, e.g., using the trait object's vtable. Because its type is unknown at compile time, trait objects are always dynamically sized types. Like all DSTs, trait objects cannot be allocated on the stack, and they need to be used behind some type of references or pointers, e.g., `&dyn MyTrait` or `Box<dyn MyTrait>`.

Here's a quick example using boxed trait objects.

```
struct Man;
struct Boy;

trait Hombre {
  fn age(&self) -> u8;
}
impl Hombre for Man {                    ①
  fn age(&self) -> u8 {
    70
  }
}
impl Hombre for Boy {
  fn age(&self) -> u8 {
    7
  }
}
fn get_age(m: Box<dyn Hombre>) -> u8 {   ②
  m.age()
}
```

① Both `Man` and `Boy` are `Hombre` in this example.

② The `get_age` function takes a `dyn Hombre` type. The smart pointer `Box` is needed since `syn Hombre` is an unsized type.

```
fn dyn_trait_demo() {
  let age = get_age(Box::new(Man));      ①
  println!("Man's age = {age}");
  let age = get_age(Box::new(Boy));      ②
  println!("Boy's age = {age}");
}
```

① This example demonstrates the *runtime polymorphism* in Rust. The `get_age` function picks the correct implementation of the `age` method based on the runtime type of the argument. In this case, the `age` will be `70`.

② In this case, `age` will be `7`.

27.2. Impl Traits

There is another situation where traits can be used in place of types. This special syntax is called the impl traits, and they are solely a static construct (although we include it in the same chapter as `dyn TRAIT`, for convenience). The `impl TRAIT` types can be used in two locations:

- As an argument type, and
- As a return type.

For example,

```
fn my_function(arg: impl MyFirstTrait) -> impl MySecondTrait {
  todo!();
}
```

The `impl` trait syntax used in the argument position is semantically equivalent to a generic function with a corresponding trait bound. For example, the following two function definitions are semantically equivalent:

```
fn my_fn<T: MyTrait>(arg: T) { }
```

```
fn my_fn(arg: impl MyTrait) { }
```

The `impl` traits in return position stand in for an (unspecified) concrete type. Each possible return value from the function implementation must resolve to the same concrete type. Note that, regardless of the actual type used in the implementation, the caller of the function can only use the methods declared by the specified `impl` trait (because that's the public interface/contract).

Here's a quick example of using `impl` traits.

```
pub trait TOne: Copy + Debug {}

impl TOne for i32 {}
impl TOne for bool {}

pub fn func_one(b: bool) -> impl TOne {  ①
  if b { 42 } else { 0 }
}
```

① The function signature (e.g., an interface) indicates that it returns `impl TOne`. In this case, that means either `i32` or `bool` (to the implementer). The actual implementation uses `i32`.

```
fn impl_trait_return_demo() {
  let r = func_one(true);                ①
  println!("{:?}", r);                   ②
}
```

① The client cannot make any assumption about the return type of `func_one`. They only know that the function returns `impl TOne`. This trait happens to include no associated items. The only public contract we know about, as a client, is that `Copy` and `Debug` are its supertraits.

② Hence, we can at least use the `:?` formatter.

```
pub trait TTwo {
  fn huh(&self) -> u8 {
    111
  }
}
impl TTwo for i32 {}
impl TTwo for bool {}

pub fn func_two(v: impl TTwo) -> u8 {     ①
  let x = v.huh();                        ②
  x
}
```

① When an impl trait is used in an argument position, it has different semantics. In this particular example, it simply means that we, as a client, can use either `i32` or `bool` to call the `func_two` function.

② In this case, the implementer, however, cannot make any assumptions about its concrete type. They can only use the public interface available through the `TTwo` trait, which happens to be the `huh` associated method, but nothing else.

```
fn impl_trait_parameter_demo() {
  let val = func_two(true);               ①
  println!("{:?}", val);
  let val = func_two(1_000_000);          ②
  println!("{:?}", val);
}
```

① The client knows that the `func_two` function takes an argument of type `impl TTwo` and that the `TTwo` trait is implemented by `i32` and `bool`. Hence, they can call this function with `bool`...

② ... Or, with `i32`.

Chapter 28. Pattern Matching

Patterns are used to match values against type variants and structures. They can also bind variables to values inside the matching structures. In Rust, pattern matching is used in:

- `let` declarations,
- `if let` expressions,
- `let else` expressions,
- `while let` expressions,
- `match` expressions,
- `for` expressions, and
- Function and closure parameters.

Note that, in particular, the `let` bindings and function and closure parameters use pattern matching, unlike in most other (imperative) programming languages. Here's a simple pattern matching example:

```
fn match_demo() {
  let legs = 2;
  match legs {                              ①
    2 => println!("A biped"),               ②
    3 | 4 => println!("Most likely, a quadruped"),
    _ => println!("Not sure what it is"),
  }
}
```

① A `match` expression.

② `2` in this example is a pattern, or more specifically, a literal pattern.

Rust supports various kinds of patterns, including

- Literal patterns,
- Range patterns,
- Wildcard patterns,
- Identifier patterns,
- Reference patterns,
- Path patterns,
- Grouped patterns,
- Rest patterns,
- Tuple patterns,
- Struct patterns,
- Slice patterns, and
- OR patterns.

Patterns can be combined and nested.

28.1. Irrefutability

A pattern that will always match a given value (from a set of all possible values) is called *irrefutable*. Irrefutable patterns should be the last arm in the `match` expression, for instance. Patterns that are not irrefutable are *refutable*, and they may, or may not, match particular values. For example,

```
if let (x, 5) = (1, 2) { }                ①
if let (x, y) = (3, 4) { }                ②
```

① The pattern `(x, 5)` is refutable. In this particular case, this pattern does not match the given value `(1, 2)`. However, if the value were `(10, 5)`, for instance, it could have matched `(x, 5)`.

② The pattern `(x, y)` is irrefutable. This pattern would match any tuple of two elements. Or, more particularly, it would match a tuple of any two integers. In this example, `x` and `y` are bound to `3` and `4`, respectively.

28.2. Destructuring

Patterns can be used to *destructure* structs, enums, and tuples. Destructuring breaks up a value into its component parts based on its structure. In a pattern of a struct, enum, or tuple type,

- A placeholder _, or the wildcard pattern, stands in for a single data field, and
- A variable-length placeholder `..`, or the rest pattern, stands in for all the remaining fields of a particular variant.

When destructuring a value with named fields, one can just write `fieldname` as a shorthand for `fieldname:fieldname`. For example, using a struct pattern,

```
struct Fruits { apple: i32, orange: i32, pear: i32 }
fn destructure() {
  let fruits = Fruits { apple: 1, orange: 2, pear: 3 };
  let Fruits {
    apple: apple_count, orange: _, pear
  } = fruits;                                ①
  println!("Apples: {apple_count}, Pears: {pear}");
}
```

① `apple_count` is bound to `apple` and `pear` is bound to `pear`. The pattern `pear` is equivalent to `pear: pear`. The `orange` field is ignored in this example.

28.3. Literal Patterns

Literal patterns use literals or literal-like values. Valid patterns are byte, string, raw string, byte string, and raw byte string literals, as well as character literals, integer literals, and integer literals preceded by a unary minus `-`. Floating point literals, and their negations, are being deprecated, and they may be disallowed in a future edition of Rust.

Literal patterns match exactly the same value as what is specified by the literal. They do not match any other values. Hence, literal patterns are always *refutable*. For instance,

```
fn literal_patterns() {
  for i in -1..=2 {                    ①
    match i {
      1 => println!("I am positive!"),   ②
      -1 => println!("I am negative!"),  ③
      _ => println!("I am nobody!"),     ④
    }
  }
}
```

① A `for` loop with a closed range `-1..=2`, which includes `-1`, `0`, `1`, and `2`, is used for illustration.

② A literal pattern `1` matches an integer literal `1`, but nothing else.

③ Likewise, a literal pattern `-1` only matches the negative integer `-1`.

④ The irrefutable wildcard pattern (`_`) matches all others. In this example, it will match `0` and `2` of the loop variable `i` in the `for` loop since `1` and `-1` are matched in the previous patterns in this `match` expression.

28.4. Range Patterns

Range patterns use the range expression-like syntax, and they are used with certain scalar types. For example, integer and character types (e.g., `u8`, `u16`, `i32`, `char`. etc.) work with range patterns. The use of floating point types (`f32` and `f64`) is being deprecated, and they should not be used moving forward.

A range pattern matches values within the range specified by their range, from a lower bound to an upper bound. An upper bound can be omitted (e.g., `l..`), in which case the range is "half-open". Otherwise (e.g., `l..=u`), it is "closed". When an upper bound is specified, the bound is included in the range.

For example,

```
fn range_patterns() {
  let c = 'm';
  let 'i'..='n' = c else {                ①
    panic!("No match!");
  };
  let i = 101;
  if let 100.. = i {                      ②
    println!("Bigger than 100: {i}");
  }
}
```

① The pattern `'i'..='n'`, used in this `let - else` expression, is a closed range pattern. It matches `'i'`, `'j'`, `'k'`, `'l'`, `'m'`, and `'n'`. Note that `..=` is a single token, and it should be written as such, e.g., without any spaces.

② The pattern `100..` of this `if let` expression is a half-open range pattern. It matches `100`, or any other valid integer number greater than `100`. Note that, in this case, `..` is part of the pattern, `100..`, whereas `=` is part of the `if let` expression syntax.

28.5. Wildcard Patterns

The underscore symbol (_) is used as a wildcard pattern, which matches any value. That is, the wildcard pattern, when used as a single pattern, is irrefutable. When it is used inside other patterns, it matches any single data field. The wildcard pattern is often used as the (last) catch-all pattern in a `match` expression. Wildcard patterns are similar to identifier patterns, but wildcard patterns are used to match values, *and ignore them*. The `_` identifier is sometimes called the discard variable. Unlike identifier patterns, wildcard patterns do not copy, move, or borrow the values they match. For example,

```
fn wildcard_patterns() {
  for pos in [Some((0, 0)), Some((3, 5)), None] {
    if let Some((x, _)) = pos {                ①
      print!("X is {x},\t");
    } else {
      print!("None,\t");
    }
    if let Some(p) = pos {
      let y = (|_: i32, y: i32| y)(p.0, p.1);  ②
      println!("Y is {y}");
    } else {
      println!("None");
    }
  }
}
```

① The second element of a tuple pattern `(x, _)` is a wildcard. The tuple pattern as a whole is irrefutable since the identifier pattern, `x`, is also irrefutable.

② The closure parameters also use patterns, and in this example, the first parameter is a wildcard. Note that this example closure implementation merely returns its second argument.

28.6. Path Patterns

Path patterns use certain path expression syntax, and they are used to refer to

- Constant values, or
- Structs or enum variants that have no fields.

```
const X: i32 = 42;
const Y: i32 = 21;

fn path_patterns() {
  let x = 42;
  match x {
    X => println!("Matched X"),          ①
    self::Y => println!("Matched Y"),    ②
    _ => println!("No match!"),
  }
  let z2 = Some::<i32>(42);
  if let None::<i32> = z2 {              ③
    println!("z2 is None::<i32>");
  }
  for z in [Some::<i32>(42), None::<i32>] {
    match z {
      Option::None::<i32> => {           ④
        println!("Matched None::<i32>");
      }
      v => {
        println!("Match any value {v:?}");
      }
    }
  }
}
```

① `X` is a path pattern that refers to the constant `X`.

② `self::Y` is also a path pattern that refers to the constant `Y`.

③ `None::<i32>` or `None` is a path pattern that refers to the constant variant of the `Option` enum.

④ The same with `Option::None::<i32>`.

28.7. Identifier Patterns

An identifier pattern binds the value they match to a variable, for example, as is most commonly used in the `let` binding. The function parameters, and closure parameters, also use identifier patterns. An identifier pattern with one identifier, mutable or immutable, is *irrefutable*. That is, the identifier pattern with one identifier matches any value and binds the value to that identifier. For example,

```
fn identifier_patterns() {
  let mut var1: f64 = 1.;                    ①
  let var2: f64 = 100.55;                    ②
  fn multiply(a: &mut f64, b: f64) -> f64 {  ③
    *a = *a * b;
    *a
  }
  let prod = multiply(&mut var1, var2);      ④
  println!("Prod = {prod}, Var1 = {var1}");
}
```

① A `let` binding with a mutable variable, `var1`.

② A `let` binding with an immutable variable, `var2`.

③ The function parameters `a` and `b` are also identifier patterns.

④ Pattern matching is attempted when the function is called.

Value-based identifier patterns have the same semantics as the `let` variable declarations. That is, the pattern binds a variable to

- A copy of the matched value if it is a `Copy` type, or
- A move from the matched value if it is a Move type.

The next section describes the reference semantics for value bindings.

28.8. Ref Identifier Patterns

The default behavior of the value-based identifier patterns, e.g., copy or move, can be changed. If the keyword `ref` or `ref mut` is used in front of an identifier, it instead binds to a reference or a mutable reference, respectively.

For instance, in the context of `let` bindings,

```
fn let_ref_bindings() {
  let ref var1: f32 = 1.0f32;            ①
  let ref mut var2: f32 = 2.0f32;        ②
  *var2 = 3.;                            ③
  println!("var1 = {var1}; var2 = {var2}");
}
```

① A `let ref` identifier pattern example. Note that the type annotation is not required. This `let ref` binding is essentially the same as `let var1: &f32 = &1.0f32`. See below, regarding the automatic `ref` binding of references to values.

② Similarly, this `let ref mut` binding is essentially the same as `let var2: &mut f32 = &mut 2.0f32`. Note that, syntactically, the variable `var2` itself is not mutable, whose type is `f32`, as explicitly annotated in this example.

③ However, `var2` is *effectively* a mutable reference.

More generally,

```
struct Car {
  make: String,
  model: String,
}

fn ref_identifier_patterns() {
  let my_car = Car {
    make: String::from("De"),
    model: String::from("Lorean"),
  };

  match my_car {
    Car {
      make: ref _make,                  ①
      model: _model,                    ②
    } => {
      println!("{}", my_car.make);      ③
      // println!("{}", my_car.model);   ④
    }
  }
}
```

① `ref _make` is a ref identifier pattern. And, therefore, the binding variable, `_make`, holds a shared reference to the `my_car.make`.

② `{model}` is an (ordinary) identifier pattern. Since `String` is a Move type, when the pattern matches, the value of `my_car.model` will be moved to `_model`.

③ Since `my_car.make` is still the owner, we can use it with no problem.

④ On the other hand, since `my_car.model` has been moved, we can no longer use it.

One thing to note is that when non-reference pattern matches a reference value, Rust uses a different binding mode. That is, the pattern is *automatically* treated as a `ref` or `ref mut` binding.

Here's another example, using the same `Car` struct example,

```
fn ref_binding_mode() {
  let your_car = Car {
    make: String::from("Trans"),
    model: String::from("Former"),
  };
  let car_or_not: Option<Car> = Some(your_car);

  if let Some(c) = &car_or_not {          ①
    println!("{}{}", c.make, c.model);    ②
    let yc = car_or_not.unwrap();         ③
    println!("{}{}", yc.make, yc.model); ④
  }
}
```

① Since the reference value `&car_or_not` is matched by a non-reference pattern `Some(c)`, the binding variable `c`, in this context, is automatically treated as a `ref` binding. That is, this `if let` expression is essentially equivalent to `if let Some(ref c) = &car_or_not { ... }`.

② Hence, `c` binds to a reference to `your_car`.

③ At this point, `car_or_not` is still the owner, and we just `unwrap` it for illustration. After this line, the value of `your_car` is moved to `yc`.

④ This `println!` statement will work without any issues.

28.9. Reference Patterns

In general, any patterns can be made into corresponding reference patterns, by using the `&` or `&&` prefix. The only exception is the range pattern.

Reference patterns dereference the references that are being matched. And hence, if any variables are bound through the match, they borrow the matched values. The `mut` reference patterns, e.g, using the `&mut` or `&&mut` prefix, can only match mutable references. As an example,

```
fn reference_patterns() {
  let a = &"Hello, Underworld!";
  let &"Hello, Underworld!" = a else {   ①
    panic!("The end of the world!");
  };

  let mut b = &Some("Hello, Overlord!");
  if let &mut &Some(x) = &mut b {        ②
    println!("x = {x}");
    return;
  };
  panic!("The end of the kingdom!");
}
```

① `&"Hello, Underworld!"` is a reference pattern based on the literal pattern `"Hello, Underworld!"`. This will match `a` in this example, and hence this `let - else` statement will not panic.

② `Some(x)` is a tuple struct pattern, as we describe a bit later in this chapter. `&mut &Some(x)` is a reference pattern that can match a reference of type `&mut &Option<&str>`. In this example, it matches successfully, and the variable `x` is bound to the string `"Hello, Overlord!"`. Note that if a pattern is *irrefutable*, then its corresponding reference pattern is also *irrefutable*.

28.10. OR Patterns

An OR pattern comprises two or more subpatterns, separated by the union symbol (`|`). An OR pattern matches if any of the subpatterns matches, by going through them from left to right. They can be nested and combined with other patterns, e.g., using grouped patterns.

Syntactically, OR patterns are allowed in any of the places where other patterns are allowed, with the exceptions of `let` bindings and function and closure arguments.

For example,

```
fn or_patterns() {
  let x: u8 = 3;
  match x {
    1 | 3 | 5 => {                      ①
      println!("Small and odd")
    }
    0 | 2..=9 => {                      ②
      println!("Small and even")
    }
    _ => println!("Big and hairy"),
  }
}
```

① This OR pattern includes three literal patterns, `1`, `3`, and `5`. In this example, `x` will not match `1`. but, it will match `3`, and hence the overall OR pattern ends up matching the value `x`.

② Another example OR pattern with a literal pattern `0` union'ed with a closed range pattern `2..=9`.

28.11. Grouped Patterns

A pattern enclosed in parentheses is also a pattern. That is, if `PAT` is a pattern, then `( PAT )` is also a pattern. Grouped patterns are primarily used to explicitly control the precedence of compound patterns. For instance, the `|` symbol in the OR patterns has a rather low precedence. Hence, grouped patterns can be used in those contexts. Here's a simple example:

```
fn grouped_patterns() {
  let a = &"ABC";
  if let &("ABC" | "XYZ") = a {
    println!("ABC or XYZ");
  }
}
```

28.12. The Rest Patterns

Rest patterns (`..`) can be used anywhere a pattern element is expected. Unlike the wildcard pattern, the rest pattern is a variable-length pattern, which can match zero, one, or more consecutive elements that are not specifically matched by other patterns. The rest pattern can appear in place of a series of pattern elements, for example, in the beginning, in the middle, or in the end, but it can be used at most once in a given pattern. Just like the wildcard pattern, the rest pattern is always irrefutable. The rest patterns can be used in

- Tuple patterns,
- Struct patterns,
- Tuple struct patterns, and
- Slice patterns.

For instance,

```
fn rest_patterns() {
  let t = (1, 2, 3, 4);
  match t {
    (1, .., 3) => println!("First and last elements are 1 and 3"),
    (1, ..) => println!("The first element is 1"),
    (.., 4) => println!("The last element is 4"),
    (..) => println!("Anything goes"),
  }
}
```

28.13. Tuple Patterns

Tuple patterns use the tuple-like syntax, comprising zero or more subpattern elements separated by commas (`,`), enclosed in parentheses. A tuple pattern matches a tuple value if each of the subpatterns matches according to their specific kinds and criteria. The tuple pattern is irrefutable if all of its subpatterns are irrefutable. It is refutable otherwise. The special tuple pattern, `(..)` with a single rest pattern enclosed in parentheses, matches a tuple of any size. The pattern `()` matches an empty tuple. For example,

```
fn tuple_patterns() {
  let t = (1, true, 'a');
  match t {
    (_, false, _) => {                    ①
      println!("The second element is false");
    }
    (1, _, 'b') => {                      ②
      println!("First element is 1. Third element is 'b'");
    }
```

```
    (2, ..) => {                          ③
      println!("The first element is 2");
    }
    (a, b, c) => {                        ④
      println!("({a}, {b}, {c})");
    }
  }
}
```

① A three-element tuple pattern. If the second element is `false`, the overall tuple pattern will match since we use the wildcard patterns for the first and the third elements.

② A similar tuple pattern. If the tuple value's first and third elements are `1` and `'b'`, respectively, the overall pattern will match.

③ This pattern only checks the first element. If it is `2`, then the value matches this tuple pattern. The rest is ignored by the rest pattern, `..`.

④ This is an irrefutable pattern since all of its subpatterns, e.g., the identifier patterns, are irrefutable. In this example, it will match any three-element tuple, or more precisely, a value of type `(i32, bool, char)`.

28.14. Struct Patterns

Struct patterns are often use to destructure values of structs or struct variants of enums. They are commonly used in a more general pattern matching context as well. A struct pattern has a form based on the struct initializer syntax, but it supports a more broader syntax, as with other kinds of patterns. Struct patterns match struct values that match all criteria defined by its subpatterns. A struct pattern is irrefutable if all of its subpattern is irrefutable.

In a struct pattern, the fields can be referenced by name, index (in the case of tuple structs), or they can be ignored by using the rest pattern (`..`).

Here's a simple of example of using struct patterns.

```
struct Point {
  x: i32,
  y: i32,
  z: i32,
}
fn struct_patterns() {
  let p = Point { x: 10, y: 10, z: 20 };
  match p {
    Point { y: 0, .. } => {              ①
      println!("I'm on the x-axis")
    }
    Point { x: 0, .. } => {              ②
      println!("I'm on the y-axis")
    }
    Point { x, y, z: _ } if x == y => {  ③
      println!("I'm diagonal: {x}")
    }
    Point { x, y, z } => {               ④
      println!("({x}, {y}, {z})")
    }
  }
}
```

① This `Point` struct pattern will match a `Point` if its field `y` has a value `0`. The rest, literally, is irrelevant for pattern matching.

② Likewise, this struct pattern will match a value if its struct field `x` has a value `0`.

③ This pattern will match as long as `x == y`, based on the match guard. The `z` value is ignored.

④ This pattern `Point {x, y, z}` is the same as `Point {x: x, y: y, z: z}`. This is an irrefutable pattern since all of its field patterns are irrefutable.

28.15. Tuple Struct Patterns

Tuple struct patterns are used to test matching tuple structs or tuple struct variants of enums. A tuple struct matches if all of its subpatterns match, corresponding to its tuple elements. They are also commonly used to destructure tuple structs or enum values, such as the `Some` variant of `Option` or the `Ok` and `Err` variants of `Result`. A tuple struct pattern is irrefutable if and only if all of its supatterns are irrefutable. For example,

```
struct Coord(f32, f32);

fn tuple_struct_patterns() {
  let c = Coord(0.1, 2.0);
  match c {
    Coord(lat, lon) if lon.abs() < 0.1 => {    ①
      println!("Latitude is {lat}. Longitude is close to 0.");
    }
    Coord(lat, lon) if lat.abs() < 0.1 => {
      println!("Latitude is close to 0. Longitude is {lon}.");
    }
    _ => println!("An arbitrary coordinate."),
  }
  let Coord(lat, lon) = c;                     ②
  println!("Latitude is {lat}. Longitude is {lon}.");
}
```

① `Coord(lat, lon)` is an irrefutable pattern since its two element subpatterns are both irrefutable. We use a match guard in this example, e.g., to match only a certain set of `Coord`'s. As indicated, in general, floating-point numbers should not be used in patterns, moving forward.

② Since `Coord(lat, lon)` is irrefutable, we simply use the `let` binding, instead of `if let`, for instance.

28.16. Slice Patterns

Syntactically, a slice pattern is a comma separated list of zero, one, or more subpatterns enclosed in square brackets (`[]`). Slice patterns are used to match various sequence values, e.g., arrays of fixed size or slices of dynamic size based on arrays or vectors. The matching criterion of a slice pattern is determined by matching of its subpatterns, much the same way that other compound patterns work. As a simple example, here's a recursive implementation of a length function.

```
fn slice_patterns() {
  let s = [1, 2, 3];
  let l = length(&s);
  println!("Length = {l}");
}

fn length<T: std::fmt::Debug>(s: &[T]) -> usize {
  match s {
    [] => 0,                                    ①
    [a,] => {                                   ②
      println!("One element: {a:?}");
      1
    }
    [_, tail @ ..] => 1 + length::<_>(tail),    ③
  }
}
```

① The empty slice pattern. It yields the length of `0`.

② This pattern is a one-element slice pattern. It is really part of the third one. We just include it here for illustration. A trailing comma is optional.

③ This is a classic "head - tail" pattern. This pattern is conceptually equivalent to `[head, ..]`. In this particular example, the actual value of `head` is not

needed, that is, for the purposes of computing the length. Each element contributes `1` to the total length. Hence we use the wildcard pattern `_` for the head. The rest pattern `..` represents the tail part (of the `&[T]` type). To be able to call the `length` function recursively with this tail part, we need a name for "the rest". `tail @ ..` is called the *at pattern*, and it is explained in the next and final section of this (looong) chapter.

28.17. At (@) Patterns

A pattern can be preceded by an identifier pattern and the `@` symbol. The overall compound pattern is usually called the *at pattern*, and it has the general syntax of `VAR @ SUBPATTERN`. If a value matches the sub-pattern, `SUBPATTERN`, then the matched value is bound to the variable, `VAR`. (In the `length` function example of the previous section, the `tail @ ..` pattern captures the matched "rest" part and binds it to the variable, `tail`.)

Here's another simple example:

```
fn at_patterns() {
  for x in [2, 10] {
    if let var1 @ 1..=5 = x {            ①
      println!("Var1 = {var1}");
    } else {
      println!("No match!");
    }
  }
}
```

① The pattern `var1 @ 1..=5` will match `2` but not `10` in this example. When the pattern matches, the matched value is bound to `var1`. Hence, when this function is called, it will print, *Var1 = 2* and *No match!* in two separate lines.

Chapter 29. Statements

Rust primarily uses expressions for program execution as well as for computing desired outputs. In most imperative programming languages, statements are the main players, and they "include" expressions, with a notable exception of function calls, which are generally expressions (and, statements). Functions in imperative languages do "include" statements.

In Rust, the role is (almost) reversed. Most forms of value-producing evaluations, with or without intentional side effects, are carried out by expressions. Expressions can be nested within other expression structures, and the rules for evaluating an expression involve specifying the order of evaluating its subexpressions as well as the value ultimately produced by the expression itself. In contrast, statements in Rust play somewhat minor roles, e.g., mostly to contain and explicitly sequence expression evaluations. (In imperative programming, a program is essentially a sequence of statements, and hence statements are "required", if you will, depending on how you define statements and expressions.) Statements generally end with semicolons (`;`), unlike expressions. But they can be omitted in some special cases.

A statement is a component of a block, which is in turn a component of an outer expression or function. Rust statements can be categorized into two classes,

- Declaration statements, and
- Expression statements.

A declaration statement is one that introduces one or more names into the enclosing statement block. The declared names may denote new variables or new items. That is, declaration statements can again be categorized into two classes,

- Item declarations, and
- `let` declarations.

29.1. Item Declarations

Items can be declared within a module, as we describe in the beginning of the book, or they can be declared within a statement block. They have a syntactically identical form. Declaring an item within a statement block restricts its scope to the block containing the declaration statement.

Nested declarations do not implicitly *capture* outer variables, such as local variables or the enclosing function's arguments or generic parameters. In particular, Rust's functions are not closures. For example,

```
fn outer(a: i32) {
  let b = 42;
  fn inner() {
    // println!("outer::a = {a}");       ①
    // println!("outer::b = {b}");       ②
  }
}
```

① The variable a from the outer block cannot be accessed here. This will cause an error.

② Likewise, this statement will cause an error.

29.2. The `let` Declarations

A `let` statement introduces a new set of variables, given by an irrefutable pattern (other than OR patterns), as we discuss, and use, throughout this book. The pattern is followed optionally by a type annotation and then optionally by an initializer expression, e.g., after the assignment operator `=`. When no type annotation is given, the compiler will infer the type, or signal an error if insufficient type information is available for definite inference.

For example,

```
fn let_demo() {
  let x: u128 = 42;                          ①
  assert_eq!(x, 42);
  let y2 = (42, 24);
  let (y, _): (i8, u8) = y2;                 ②
  assert_eq!(y, 42);
  let w2 = [1, 1, 2, 3, 42, 8, 13];
  let [.., w, _, _]: [i32; 7] = w2;          ③
  assert_eq!(w, 42);
  struct S { p1: i8, _p2: i8 }
  let z2 = S { p1: 42, _p2: 84 };
  let S { p1: z, _p2: _ }: S = z2;           ④
  assert_eq!(z, 42);
  enum E { V(i32) }
  let e2 = E::V(42);
  let E::V(u) = e2;                          ⑤
  assert_eq!(u, 42);
}
```

① A simple `let` declaration. The variable on the left hand side is the irrefutable identifier pattern.

② A `let` declaration using a tuple pattern. This is often known as destructuring. One can use multiple variables on the left hand side.

③ Another example, using a slice pattern. Note that, in rust, destructuring is just a special case of broader pattern-based `let` binding syntax.

④ One can also use a struct pattern for declaring new variables.

⑤ Likewise, this `let` binding uses a path pattern, e.g., for an enum variant. Note that, any irrefutable pattern, other than ORs, can be used on the left hand side of the `let` declaration statement.

Variables introduced by `let` declarations are visible from the point of declaration until the end of the enclosing block, or until they are shadowed by the same variable declaration.

29.3. Expression Statements

Rust includes a number of different kinds of expressions, and all of them can be used as statements. When an expression is used syntactically as a statement, it is called the *expression statement*. An expression statement *evaluates* its expression and then *ignores* the result. As a general rule, an expression statement's purpose is to trigger any side effects of evaluating its expression.

An expression that does not end with a block needs a semicolon at the end to be a valid statement. An expression that consists of only a block expression or control flow expression, whose result type is `()`, can omit the trailing semicolon, if it is used in a context where a statement is expected. For example,

```
fn expression_statements(a: i32) {
  42;                                    ①
  if a > 42 { true } else { false };     ②
  { println!("Forty two!!!") }           ③
}
```

① Although it is a valid expression statement (with the trailing `;`), it is pretty much of no use since the value `42` has *no* side effect.

② Ditto. Note that the semicolon is required since the type of the `if` expression in this example is `bool`, and not `()`.

③ In this case, the trailing `;` is optional since the type of the block expression, which is the type of the `println!` macro call, is `()`. This expression statement clearly has a side effect, that is, printing out the string value to the console.

Chapter 30. Simple Expressions

Rust is primarily an expression-based language, similar to many (pure) functional programming languages. Unlike in functional languages, however, expressions in Rust can have *side effects* (sometimes, just called "effects"). In Rust, an expression therefore plays two roles:

- It always evaluates to a value (including `( )`), and
- It may have *effects* during evaluation, which can affect the program execution (in a similar way that statements do in imperative programming languages).

In fact, as we discuss in the previous chapter, everything in Rust is an expression (and, a statement), that is, with the exception of declaration statements.

Although it is hard to define precisely what functional programming (FP) is, one of the most important ingredients of the languages that support *FP* well is their focus on expressions. Support for pure functions and immutability, etc., matters only when you can support full expressiveness via expressions first. In this sense, Rust is one of the best (imperative) languages for FP. Much of Rust code is written, at least syntactically, in FP styles. Whether it is really FP or not is debateable, as we indicated earlier, since Rust's expressions (can) have *effects* unlike those of the pure FP languages. But, regardless, the idiomatic programming style in Rust is rather different from that of more traditional imperative languages. Some people call Rust the "gateway drug to functional programming". ☺

At any rate, if you are new to Rust, and if you are coming from "pure imperative" languages like (©) or Go, then we cannot overemphasize the importance of the expression-oriented programming styles in Rust. Note that one of the most important characteristics of the *expressions* is that they return a value (regardless of whether they have side effects or not). On the other hand, the statements do not have values. Or, their values are always `( )` (or, `!`).

Expressions often contain sub-expressions, called the operands of the expression. The meaning of each kind of expression dictates several things:

- Whether or not to evaluate the operands when evaluating the expression,
- The order in which to evaluate the operands, and
- How to combine the operands' values to obtain the value of the expression.

In this way, the structure of expressions dictates the structure of execution. Blocks are expressions in Rust, and therefore blocks, statements, expressions, and blocks again can recursively nest inside each other to an arbitrary depth.

30.1. Literal Expressions

The literals that we describe earlier in the lexical analysis chapter are expressions by themselves. Their expression values are the values represented by the literals.

- Boolean literals,
- Integer literals,
- Floating-point literals,
- Character literals,
- String literals,
- Raw string literals,
- Byte literals,
- Byte string literals, and
- Raw byte string literals.

30.2. Path Expressions

A path, when used in an expression context, refers to an item or a local variable. They are considered simple expressions.

30.3. Grouped Expressions

A grouped expressions, or parenthesized expression, comprising,

- An opening parenthesis `(`,
- An enclosed operand expression, and
- A closing parenthesis `)`,

evaluates to the value of the enclosed expression. Generally, parenthesized expressions are used to affect the evaluation order of the operand sub-expressions within an expression, or otherwise to make the expressions easier to read.

For example,

```
fn grouped_expressions() {
  let a = 1 + 2 * 3;                    ①
  let b = (1 + 2) * 3;                  ②

  assert_eq!(a, 7);
  assert_eq!(b, 9);
}
```

① Since the multiplication has a higher precedence than the addition, the sub-expression `2 * 3` is evaluated first.

② The explicitly grouped sub-expression `(1 + 2)` is evaluated first in this case, resulting in 9, say, instead of 7.

Here's another example:

```
struct Factory<'a> {
  ctor: fn(&'a str) -> String,
}

fn function_pointer() {
  let fac = Factory { ctor: String::from };
  let ferris = (fac.ctor)("Ferris");     ①
  println!("Ferris = {ferris}");
}
```

① The function call expression has a higher precedence than the struct member access. Hence, in this example, the expression `(fac.ctor)` should be explicitly grouped. Otherwise, it is an invalid expression.

30.4. Operators

Operators are defined for the builtin types by the Rust language. Many of the operators can also be *overloaded* using the traits in the `std::ops` and `std::cmp` modules. They are further discussed later.

30.5. The `await` Expressions

The `await` operator can be applied to an expression that returns a `Future` (defined in the `std::future` module), and it suspends the current computation until the given future is ready to produce a value. We do not include async-await programming in this book.

Chapter 31. Call Expressions

31.1. Function Calls

A call expression *calls* a function. The syntax of a call expression is

- A (callable) expression, called the *function operand*, followed by
- A comma-separated list of expressions in a pair of parentheses, called the *argument operands*.

If, and when, the call ultimately returns, the expression completes. In general, in an expression of the form `f(...)`, if `f` is not an inherent function type then the expression uses the method defined in one of the following three traits. (As indicated, these traits are primarily used for closures, and you cannot currently implement these traits on your types in stable Rust. But, this might change in the near future.)

- `std::ops::Fn`,
- `std::ops::FnMut`, or
- `std::ops::FnOnce`.

They differ in whether they take an instance of the type

- By shared reference,
- By mutable reference, or
- By value, respectively.

As mentioned earlier, function parameters are essentially local variables, and the function call binds these parameters to the passed-in arguments, e.g., by immutably or mutably borrowing, or by copying or moving, the values.

31.2. Method Calls

A *method call* consists of

- An expression (called the *receiver*), followed by
- A dot (`.`),
- A (callable) expression path segment, and
- A comma-separated list of argument expressions in parentheses.

When resolving a method call, the receiver may be automatically dereferenced or borrowed in order to call the method. Since there are syntactically four different ways to declare a receiver, e.g., `self`, `mut self`, `&self`, and `&mut self`, the method call expressions follow the similar pattern. For example,

```
#[derive(Debug)] struct S(u8);
impl S {
  fn f1(self) {
    println!("{self:?}");
  }
  fn f2(mut self) {
    self.0 += 1;
    println!("{self:?}");
  }
  fn f3(&self) {
    println!("{self:?}");
  }
  fn f4(&mut self) {
    self.0 *= 2;
    println!("{self:?}");
  }
}
```

```
fn method_calls() {
  let s1 = S(0); s1.f1();                ①
  // println!("{s1:?}");                 ②
  let s2 = S(10); s2.f2();               ③
  // println!("{s2:?}")
  let s3 = S(20); s3.f3();               ④
  println!("{s3:?}");                    ⑤
  let mut s4 = S(30); s4.f4();           ⑥
  println!("{s4:?}");
}
```

① This method call `s1.f1()` moves `s1` into `f1`.

② `s1` can no longer be used since it has been moved.

③ This call moves `s2` into `f2`. Note that `s2` does not need to be mutable.

④ Calling `s3.f3()` automatically borrows `s3`.

⑤ No issue using `s3` after the borrow is done.

⑥ `s4` needs to be mutable to be able to use a method that mutably borrows the receiver. A mutable borrow happens when we call `s4.f4()`.

31.3. The `return` Expressions

Rust's `return` expressions work more or less the same way, but unlike in many other programming languages, they are expressions, and not statements.

When the `return` expression includes an operand after the keyword `return`, the value of the `return` expression is the value of this operand. Otherwise, its value the unit value `()`. The value of the function call expression is the value of the `return` expression that is evaluated before the control flow returns to its caller, that is, if such a `return` expression exists.

For example,

```
fn return_expressions() {
  fn returns_nothing() {
    return;                                  ①
  }
  let r1 = returns_nothing();                ②
  println!("r1 = {r1:?}");

  fn returns_something() -> u128 {
    return 42;                               ③
  }
  let r2 = returns_something();
  println!("r2 = {r2}");                     ④
}
```

① The `return` expression is often used as a statement. This statement could have been written as expressions, `return`, `return ()`, or just `()` (without the trailing semicolons).

② The type and value of `r1` are `()` and `()`, respectively.

③ Ditto. `return` expressions are almost always used in a statement context and they usually take the form of the statement. But, in cases like this, it is more common to just use the return value, `42`, as the last expression of the function body block. The `return` expression/statement is required when the function needs to return the control to the caller before it reaches the end of the block.

④ The type and value of `r2` are `u128` and `42`, respectively.

Chapter 32. Expressions with Blocks

The following expressions include blocks as part of their syntax.

- Block expressions,
- Async block expressions,
- If expressions,
- If let expressions,
- Match expressions, and
- Various loop expressions.

Loop expressions are discussed in the next chapter.

32.1. The Block Expressions

A block expression, or block (`{...}`), is a control flow expression. The block also serves as an anonymous namespace scope for the enclosed item and variable declarations.

The syntax for a block is

- An opening brace, `{`, which can be preceded by an optional label,
- Any optional inner attributes,
- Zero, one, or more statements,
- An optional *final operand*, and
- A matching closing brace, `}`.

As a control flow expression

A block sequentially executes its enclosed statements, and then it evaluates its final operand expression, if any.

As an anonymous namespace scope

The usual block scoping rules apply. The items declared within the block are in scope inside the block, and `let`-bound variables are in scope from the statement following the declaration until the end of the block.

It's worth emphasizing that, unlike in many C-style and other imperative programming languages, a block in Rust is an expression (which can include other statements), and not a statement. This implies, among other things, that *anything* in Rust can be converted to an expression, e.g., just by enclosing it in a block. This gives Rust the *super power*, if you will, e.g., the expressiveness of functional programming languages.

When evaluating a block expression, each statement is executed sequentially. Then the optional final operand is evaluated. The value and type of a block is

- The value and type of the `break` expression, if one is used to exit the block,
- The value and type of the final operand, if one is present, or
- `()` and `()`, respectively, otherwise.

We include various example uses of the block expressions throughout the book.

32.2. The `async` Block Expressions

An `async` block is a variant of a block expression, and it evaluates to a `Future`. `await` expressions can be included in an `async` block. The final expression of the block, if present, determines the result value of the future.

32.3. The `match` Expressions

A `match` expression uses pattern matching to select a value from one or more branches, or "arms", which are enclosed in a block. Rust supports a few different kinds of expressions that use patterns, but the `match` expression is the granddaddy of them all. Other forms of pattern expressions may be considered a syntactic shorthand of `match` expressions. Here's a common syntax:

```
match SCRUTINEE-EXP {
  PAT1 => EXP1,
  ...
  PATn => EXPn,
}
```

The value of `SCRUTINEE-EXP` is compared to the series of patterns, denoted `PAT1` ... `PATn` in this notation with `n >= 1`, from top to bottom. Then, the corresponding value to the matched pattern becomes the value of the overall `match` expression. A few things to note:

- The `EXP`'s can be a block expression, in which case, the trailing comma can be omitted. The trailing comma for the last pattern is optional.
- The set of patterns needs to be "exhaustive", that is, they have to cover all possible values of `SCRUTINEE-EXP`.
- All patterns should match at least one possible value of `SCRUTINEE-EXP`.
- The types of all patterns should be the same, and they must be equal to that of `SCRUTINEE-EXP`.
- The types of all expressions on the right hand side, `EXP1` ... `EXPn`, must be the same, and this type is the type of the overall `match` expression.
- The value of the `match` expression is that of the first successful match.

For example,

```
fn match_expressions() {
  for rank in &['A', 'J', 'K', 'X'] {
    let value: Vec<u32> = match rank {    ①
      'A' => vec![1, 11],                 ②
      'K' | 'Q' | 'J' | 'T' => vec![10], ③
      r @ ('2'..='9') => {                ④
        vec![r.to_digit(10).unwrap()]
      }
      _ => panic!("Invalid rank {rank}") ⑤
    };                                    ⑥
    println!("Rank: {rank}, Value: {value:?}");
  }
}
```

① The type of `rank` is `char`. The type of the overall `match` expression is `Vec<u32>`, whose value is bound to a local variable `value` in this example.

② The pattern, `'A'`, is a literal pattern. If, and only if, the value of `rank` is `'A'`, it will match, and it will return a value `vec![1, 11]`. Note that `Vec<T>` is a Move type, and hence `value` will take over this vector value when this `match` expression returns.

③ An OR pattern. As described in the Patterns chapter, the sub-patterns in the OR pattern are tested from left to right. As you might have noticed, this `match` expression evaluates the card value(s) of a given card in the game of blackjack.

④ This is an `@` pattern, with a range subpattern. If `rank` happens to be one of the values from `2` to `9`, then the matched value is bound to `r`, which is then used to derive the card value.

⑤ A wildcard pattern.

⑥ As indicated, at the end of this `match` block, the matched value is moved out of this block, which is bound to the variable `value` in this example.

32.3.1. Match guards

In general, a pattern in each *arm* may be followed by the `if` keyword and a boolean expression. They are called the *match guards*. Each subpattern of an OR pattern can include a separate match guard. If a (sub-)pattern matches, and if it has a match guard, then it is evaluated next, and if and only if this evaluates to `true`, the overall (sub-)pattern is considered a *match*.

Note that the scope of any variables introduced in the pattern, e.g., through the identifier patterns or the at patterns, includes the match guard, if any, and the following expression.

```
fn match_guards() {
  let p = (1., 0.);
  match p {
    (x, y) if x == 0. && y == 0. => {     ①
      println!("I'm the origin");
    }
    (x, y) if radius(x, y) == 1. => {     ②
      println!("I'm on a unit circle");
    }
    (x, y) if x == -y => {                ③
      println!(r#"I'm on an "anti-diagonal""#);
    }
    (x, y) => {                           ④
      println!("I'm somewhere, ({x}, {y})");
    }
  }

  fn radius(x: f32, y: f32) -> f32 {
    (x.powf(2.) + y.powf(2.)).sqrt()
  }
}
```

① As stated, use of floating point numbers in the patterns is discouraged. This tuple pattern with a match guard is effectively the same as `(0., 0.)`.

② A different match guard example.

③ Another match guard example. Note that the pattern `(x, y)` is irrefutable by itself, and the match guards act like `if else` branching in this example.

④ The irrefutable pattern `(x, y)` is being used as the last catch-all pattern, thereby making the set of patterns *exhaustive* over all 2-tuples of `(f32, f32)`.

32.4. The `if` Expressions

Rust currently supports three kinds of *conditional block expressions*, `if - else`, `if let - else`, and `let - else`. An `if` expression is conditionally evaluated based on a condition operand, which evaluates to a `bool` value. The `if - else` expression comprises

- An `if` block, that is,
  - The `if` keyword, a condition expression, and a block, followed by
- An optional trailing `else` block, that is,
  - The `else` keyword, and a block.

The `if` block is a block expression. The `else` block can also be a block expression, or it can be instead other `if` or `if let` expression. Hence, for example, using our informal notation, the syntax can be represented as follows:

```
if CONDITION { /*...*/ }
if CONDITION { /*...*/ } else { /*...*/ }
if CONDITION { /*...*/ } else if COND1 { /*...*/ }
if CONDITION { /*...*/ } else if COND1 { /*...*/ } else { /*...*/ }
// etc. ...
```

Note that there can be any number of `else if CONDITION { /* ... */ }` blocks. All blocks must have the same type, which can be `()`, or even `!`. This is the type of the overall `if` expression.

The condition operands are sequentially evaluated from the beginning (e.g., the operand associated with the `if` block). If any condition operand evaluates to `true`, the associated block is evaluated and any subsequent `else if` or `else` block is skipped. If none of the conditions evaluate to `true`, then the `else` block, if any, is evaluated.

Note that when there is a possibility that no blocks can be evaluated, e.g., because there is no `else` block, etc., the type of each block, and hence that of the overall `if` expression, should be `()`,

Here's a simple example:

```
fn if_else_expressions() {
  let today = 30;
  let sign = if today >= 42 {            ①
    333                                  ②
  } else {
    666
  };                                     ③

  if sign >= 999 {                       ④
    println!("Ominous");                 ⑤
  } else if sign >= 666 {
    println!("Not so lucky");
  } else {
    println!("Weird");
  }                                      ⑥
}
```

① The `if` expression is an *expression*, and it returns a value.

② The type of this `if` expression is `i32`.

③ The value of this expression is bound to a variable `sign` through a `let` variable declaration statement.

④ This `if` expression is being used as a statement.

⑤ The value of the `println!` macro call is `()`.

⑥ The trailing semicolon is optional in this case.

32.5. The `if let` Expressions

Unlike `if`, which branches on boolean operands, the `if let` expression (and, `let - else` as well) uses pattern matching. Otherwise, `if let - else` has a similar syntax and semantics to `if - else`.

- An `if let` block, that is,
  - The keywords `if` and `let`,
  - A refutable pattern,
  - An equal sign `=`,
  - A scrutinee expression, as in `match` expressions, and
  - A block, followed by
- An optional trailing `else` block, e.g.,
  - The `else` keyword, and
  - A block.

As with the `if` expression, the block following `if let PATTERN = EXPRESSION` is a block expression, and the optional `else` block can be one of a block expression, e.g., `if - else` or `if let - else` expressions. Hence, just like `if -`

`else if ... else` expressions, `if let PAT = EXP - else` expressions can be chained, for example, as `if let PAT1 = EXP1 ... else if let PAT2 = EXP2 ... else ...`, etc., with any number of `else if let` blocks.

If the value of the scrutinee matches the pattern, then the corresponding block will be evaluated. Like `if` expressions, all blocks in `if let` expressions must have the same type, and the value of an overall `if let` expression is the value of the block that is evaluated. For example,

```
#[derive(Debug, PartialEq)]
enum Greek {
  Alpha(u8),
  Beta(u8),
  Omega,
}

fn if_let_expressions() {
  use Greek::*;
  let a = Alpha(42);

  if let Alpha(21) = a {
    println!("a is Alpha(21)");
  } else if let Beta(x) = a {
    println!("a is Beta({x})");
  } else if let Alpha(y) = a {          ①
    println!("a is Alpha({y})");
  } else {
    println!("a is no Greek");
  }
}
```

① This pattern `Alpha(y)` ends up matching the value `a` in this example. The variable `y` is then bound to the value `42`.

Since the block following the `else` in either `if` or `if let` expression can be either of them, the conditional `if` and pattern matching `if let` expressions can be, in fact, intermixed.

```
fn ifs_and_if_lets() {
  use Greek::*;
  let value = Beta(84);

  let x = if let Alpha(_) = value {      ①
    "1000"
  } else if Beta(42) == value {          ②
    "100"
  } else if let Omega = value {          ③
    "10"
  } else {
    "0"
  };
  println!("x = {x}");
}
```

① The rhs side of this `let` declaration is one `if let` expression that extends until the semicolon at the end of the declaration.

② It includes an `if - else` branch.

③ It also includes another `if let - else` branch.

The `if let` expression is a syntactic shorthand for a certain `match` expression. For example,

```
if let PATT = EXPR {
  STATEMENTS
}
```

This `if let` statement is equivalent to the following:

```
match EXPR {
  PATT => {
    STATEMENTS
  }
  _ => (),
}
```

Or, more generally, the following two statements are more or less equivalent to each other:

```
let x = if let PATT = EXPR {
  /* if-let block */
} else {
  /* else block */
};
```

```
let x = match EXPR {
  PATT => {
    /* if-let block */
  }
  _ => {
    /* else block */
  }
};
```

Here, `PATT` and `EXPR` represent a refutable pattern and an expression, respectively.

Note that the `let` binding declaration and the `if let` expression have some similarities, but they are also rather different from each other.

- The lhs pattern in the `let` binding must be *irrefutable*, whereas the pattern of the `if let` expression should be normally *refutable*.
- The OR pattern is not allowed in the `let` binding.

Here's a simple example of the OR patterns used in the `if let` expression.

```
fn if_let_or_patterns() {
  use Greek::*;
  let a = Alpha(42);

  if let Alpha(21) | Beta(21) = a {      ①
    println!("a contains 21");
  } else if let Alpha(x) | Beta(x) = a { ②
    println!("a contains {x}");
  }
}
```

① Using the same `Greek` enum from earlier.

② This OR pattern is still not irrefutable because of the missing `Unknown` variant. Note that OR patterns in the `if let` context must include the same variable(s) in each sub-pattern.

32.6. The `let - else` Expressions

As of version 1.65, the `let - else` expression is part of the stable Rust. This covers the "second half" of the `if let` expression, if you will. That is, the primary block of the `lf let` expression is executed when the pattern matches a scrutinee expression. On the other hand, the primary block of the `let - else` expression is executed *when the match fails*.

```
fn let_else_demo() {
  let food = Some("cookies");
  let Some(x) = food else {                ①
    panic!("No cookies!!!");               ②
  };                                       ③

  println!("Finally, some {x}!");          ④
}
```

① As with `if let`, the pattern in `let - else` should generally be refutable.

② It is a common idiom to just panic, or branch out, when the pattern does not match in `let - else`.

③ The `let - else` expression syntactically requires the trailing semicolon.

④ Note the scope of the variable `x` used in the `let - else` pattern. It extends beyond the `let - else` statement. Although the pattern `Some(x)` itself is refutable, after the `else` branching, it is safe to use the bound variable `x` at this point. `let - else` is rather similar to the simple `let` binding in this regards.

A `let - else` expression is a syntactic shorthand for a certain `match` expression. For example, the following two code snippets are more or less equivalent.

```
let PATT = EXPR else { PANIC }
STATEMENTS
```

```
match EXPR {
  PATT => { STATEMENTS }
  _ => { PANIC }
}
```

Chapter 33. Loop Expressions

Rust supports four different kinds of loop expressions:

- A `loop` expression loops indefinitely.
- A `for` expression loops over an iterable.
- A `while` expression loops as long as a boolean expression evaluates to `true`.
- A `while let` expression loops as long as a pattern match succeeds.

All four types of loops can be declared with labels, and they all support `break` and `continue` expressions.

33.1. The `loop` Expression

A `loop` expression consists of the keyword `loop` followed by a block expression. It repeatedly executes the block expression, which amounts to an infinite loop. And, hence its type may be never `!`. A `loop` expression that terminates via a `break` has a type compatible with the value of that `break` expression.

```
use std::thread::sleep;
use std::time::Duration;

fn loop_loop() -> ! {
  loop {                                    ①
    sleep(Duration::from_millis(1000));
    println!("Slept another second");
  }
}
```

① The type of this expression is `!`, and it is used as a statement here.

33.2. The `for` Expression

A `for` expression is used to loop over a sequence of an `IntoIterator` type. The `IntoIterator` trait defines a method, `into_iter`, which returns an `Iterator` type. `Iterator` defines one required method, `next`, which is used to get the next element in a sequence.

If the iterator yields a value, that value is matched against the irrefutable pattern of the `for` expression, and if successful, the body of the loop is executed. Then, control returns to the head of the `for` loop to repeat the iteration. When the iterator is empty, the `for` expression completes.

For example,

```
fn for_loops() {
  let arr1 = ['C', 'a', 'T'];              ①
  for a in arr1 {                          ②
    println!("{a}");
  }

  let arr2 = vec!["Hi", "World"];          ③
  for a in &arr2 {                         ④
    println!("{a}");
  }

  let mut arr3 = [1, 2, 3, 5, 8];
  for a in &mut arr3 {                     ⑤
    *a += 1;
    println!("{a}");
  }
  println!("{arr3:?}");                    ⑥
}
```

① The builtin array types implement `IntoIterator`.

② Iterating over each value of `arr1`. The type of the *loop variable* `a` is `char`. This expression is equivalent to `for a in arr1.into_iter() { ... }`. The `into_iter` method returns an iterator, whose `next` method is then called to get the next element in the given iterator, e.g., wrapped in an `Option`. If it is a `Some(T)` value, it is automatically unwrapped. Otherwise, that is, if `None` is returned, then the iteration terminates. Rust automatically handles all this for `for` expressions. Normally, if you need to implement your own `IntoIterator` type, and custom iteration logic, then this needs to be all manually taken care of.

③ The `Vec` type also implements `IntoIterator`.

④ Iterating over borrowed references. The type of `a` is `&(&str)`.

⑤ Iterating over mutable references. The type of `a` is `&mut i32`.

⑥ This will print *[2, 3, 4, 6, 9]*.

Rust's range expressions also implement the `IntoIterator` trait. They are sometimes used just like C's classic `for` loop.

```
fn for_range_loops() {
  for i in 5..=8 {                    ①
    println!("{i}");
  }
  for c in 'c'..'t' {                 ②
    println!("{c}");
  }
}
```

① This `for` loop goes through four `i32` elements, `5`, `6`, `7`, and `8`.

② Likewise, this iterates over 17 `char` elements.

33.3. The `while` Expressions

Rust's `while` loop is rather similar to those found in other C-style languages. The main difference is again that it is an expression in Rust.

A `while` loop begins by evaluating the boolean loop conditional operand. If the loop conditional operand evaluates to `true`, the loop body block executes, and then control returns to the loop conditional operand. If the loop conditional expression evaluates to `false`, the `while` expression completes.

For example,

```
fn while_loops() {
  let (mut i, mut j) = (0, 0);

  while i + j < 5 {                       ①
    println!("Counting: {}", i + j);
    (i, j) = (i + 1, j + 1);
  }
  println!("Done: i = {i}, j = {j}");     ②
}
```

① The value of this `while` expression is the unit value, `()`, and it is ignored. That is, `while` is used as a statement in this example, as is generally the case.

② This will output *Done: i = 3, j = 3*.

33.4. The `while let` Expressions

The `while let` expression is a variation of the `while` expression. They are semantically rather similar, but `while let` uses pattern matching rather than conditional expressions as in `while`.

More specifically, the `while let` expression consists of

- The keywords `while` and `let`, followed by
- A (refutable) pattern,
- An equal sign `=`,
- A scrutinee expression, and
- A block expression.

If the value of the scrutinee matches the `while let` pattern, the loop body block executes, and then control returns to the pattern matching clause. Otherwise, the `while let` expression completes.

```
fn while_let_loops() {
  use std::time::{SystemTime, UNIX_EPOCH};

  fn random() -> Option<i32> {           ①
    if let Ok(s) = SystemTime::now().duration_since(UNIX_EPOCH) {
      if s.as_micros() % 2 == 0 {
        Some(1)
      } else {
        None
      }
    } else {
      None
    }
  }

  while let Some(i) = random() {         ②
    println!("{i}");                     ③
  }
}
```

① *Pay no attention to that man behind the curtain.* Just an arbitrary code.

② The (main) `while let` example. Roughly half of the time, the `Some(i)` pattern will match. Once it fails to match, the loop will terminate. Note that the pattern needs to be refutable. Otherwise, it will become an infinite loop. The value of this `while let` expression is again `()`.

③ This can potentially print zero or more lines of `1`.

Multiple patterns may be specified with the `|` operator. (E.g, the OR pattern.) This has the same semantics as with `|` in `match` expressions:

```
#[derive(Debug)]
enum Degree {                                   ①
  Unknown,
  Celsius(f32),
  Fahrenheit(f32),
}

fn while_let_or_patterns() {
  use Degree::*;
  let mut degrees = vec![Celsius(100.), Unknown, Fahrenheit(37.)];
  println!("{degrees:?}");
  while let Some(Celsius(t)) | Some(Fahrenheit(t))   ②
    = degrees.pop() {
    println!("t = {t} C/F");                ③
  }
}
```

① Pattern matching over enums is a rather common *idiom* in Rust.

② The pattern has to be refutable. In this example, the pattern does not match `Degree::Unknown`, and hence it is not irrefutable.

③ This will output *t = 37 C/F*.

33.5. Loop Labels

Any of the loop expressions, `loop`, `for`, `while`, or `while let`, as well as block expressions (as of Rust 1.65 and later), may optionally be prefixed with a label. Labels are lexically more or less the same as lifetime parameters. That is, they are identifiers preceded by an apostrophe (`'`). Loop labels are followed by a colon (`:`) and a loop or block expression. For example,

```
fn loop_labels() {
  #![allow(unused, while_true, irrefutable_let_patterns)]
  'loop1: loop {}
  'loop2: while true {}
  'loop3: while let age = 42 {}
  'loop4: for i in [1, 2, 3] {}
  'block5: { print!("Empty") }
}
```

If a loop label is defined, then the nested labeled `break` and `continue` expressions may exit out of this loop or return control to its head. Rust's `break` and `continue` expressions are described next.

33.6. The `break` Expressions

A `break` expression is only allowed inside a loop or block expression. It can be one of the following four forms:

- `break`,
- `break 'LABEL`,
- `break EXPR`, or
- `break 'LABEL EXPR`.

Note the last two forms are only allowed in `loop` or block expressions.

Normally, when `break` is encountered, execution of the innermost loop, which is enclosing the `break` expression, is terminated.

```
fn break_expressions_1() {
  let mut looped = 0;

  while looped < 42 {
    loop {
      looped += 10;
      if looped > 21 {
        break;                          ①
      }
    }
    println!("{looped}");               ②
  }
}
```

① This `break` expression statement will break out of the inner `loop` when `looped` becomes bigger than `21`, e.g., `30` and upward in this example.

② This will end up printing three lines, *30*, *40*, and *50*.

This behavior can be controlled using loop labels. For example,

```
fn break_expressions_2() {
  let (mut x, mut y) = ('\0', 0u8);

  'fst: for i in 'a'..='z' {
    '_snd: for j in 1..=100 {
      if i != 'i' {
        break;                          ①
      }
```

```
      if j == 21 {
        x = ((i as u8) - 32) as char;
        y = j + 21;
        break 'fst;                          ②
      }
    }
  }
  println!("x = {x}, y = {y}");
}
```

① This `break` will only break out of the inner `for` (labeled with an (unused) label, `'_snd`).

② This will break out of all loops which are nested within the `for` loop with the label `'fst`, including the loop itself.

33.6.1. `break` and `loop` values

A `break` expression may be used to return a value from its enclosing `loop` or block expression, or from a labeled `loop` or block. One can use one of the forms `break EXPR` or `break 'label EXPR`, where `EXPR` is an expression whose result is returned from the loop/block. For example,

```
fn break_expressions_3() {
  let z = 'x: loop {
    loop {
      break 'x 333;
    }
  };
  println!("z = {z}");                        ①
}
```

① This will output *z = 333*. The type of `z` is `i32`.

33.7. The `continue` Expressions

The `continue` expression has two forms:

- `continue`, or
- `continue 'LABEL`.

They can be used only in a loop, and when the first form of `continue` is encountered, it immediately returns control to the head of the innermost loop. When a loop is associated with a label, the `continue 'LABEL` expression returns control to the head of that loop, possibly skipping any nested loops. For example,

```
fn continue_expressions() {
  let mut x = 0;
  'w1: while { x += 1; x } < 10 {          ①
    let mut y = 0;
    while { y += 1; y } < 10 {
      let sum = x + y;
      if sum % 3 == 0 {
        continue 'w1;                      ②
      } else if sum % 2 == 0 {
        continue;                          ③
      }
      println!("sum = {sum}");
    }
  }
}
```

① *With great power comes great responsibility.* For illustration only. ☺

② It continues the outer loop, with the current value of the loop variable `x`.

③ It continues the inner loop, without changing the values of `x` and `y`.

Chapter 34. Operators

Rust includes most of the operators commonly found in other programming languages, and possibly more. One exception is probably C's increment and decrement operators, which have recently become, for some reason, villains in programming. The rationale behind this new aversion to these operators is that they are too difficult, and too confusing, to use for *common average developers*. And yet, these modern programming languages, including Go and Rust, include features that are ten times or a hundred times more complex. ☺

As stated, *everything* is an expression in Rust. But, that *everything* excludes declarations, and assignment, for all practical purposes. If you are used to using increment and decrement operators, you may (or, may not) miss them. Regardless, however, Rust's expressive power is still literally *through the roof* if you compare it with other imperative programming languages. ☺

We go through some of the common operators *very briefly*, in this chapter, mainly for completeness, but their syntax and semantics are pretty much the same as those found in other C-style languages. We discuss operator overloading and related operator traits in the following chapter, Operator Overloading.

34.1. Unary Operators

The unary negation operator (`-`) changes the sign of a given numeric operand. This operator can be overloaded by implementing the `std::ops::Neg` trait. For example,

```
fn negation_operator() {
  let (x, y, z) = (0, 10, 22.2);
  assert_eq!((-x, -y, -z), (0, -10, -22.2));
}
```

The unary NOT operator (`!`), which can be overloaded by implementing the `std::ops::Not` trait, is commonly used for,

- A logical NOT operation for a `bool` operand, which flips the logical value, or
- A bitwise NOT operator for a numeric operand, which reverses all bits in a given operand.

```
fn logical_not_operator() {
  let (b1, b2) = (true, false);
  assert_eq!((!b1, !!b1, !b2, !!b2), (false, true, true, false));
}

fn bitwise_not_operator() {
  let u1 = 0b01001u8;
  println!(
    "u1 \-> !u1 = \
    {0:b} ({0}) -> {1:b} ({1})",
    u1, !u1
  );                                          ①

  let i1 = 0b01001i8;
  println!(
    "i1 -> !i1 = \
    {0:b} ({0}) -> {1:b} ({1})",
    i1, !i1
  );                                          ②
}
```

① The output: *u1 -> !u1 = 1001 (9) -> 11110110 (246)*.

② The output: *i1 -> !i1 = 1001 (9) -> 11110110 (-10)*.

34.2. Arithmetic Binary Operators

Symbol	Name	Overloading Trait
`+`	Addition	`std::ops::Add`
`-`	Subtraction	`std::ops::Sub`
`*`	Multiplication	`std::ops::Mul`
`/`	Division	`std::ops::Div`
`%`	Remainder	`std::ops::Rem`

Binary operator expressions are all written with infix notation, e.g., `lhs OP rhs`. Here are some examples:

```
fn arithmetic_demo() {
  println!("5 + 2 = {a}", a = 5 + 2);
  println!("5 * 2 = {m}", m = 5 * 2);
  println!("5 / 2 = {d}", d = 5 / 2);     ①
  println!("-5 / 2 = {d}", d = -5 / 2);   ②
  println!("5 % 2 = {r}", r = 5 % 2);     ③
  println!("5 % -2 = {r}", r = 5 % -2);   ④
  println!("-5 % 2 = {r}", r = -5 % 2);   ⑤
}
```

① The output: *5 / 2 = 2*.

② The output: *-5 / 2 = -2*. Note that integer division rounds towards zero.

③ The output: *5 % 2 = 1*.

④ The output: *5 % -2 = 1*.

⑤ The output: *-5 % 2 = -1*. Note that the remainder has the same sign as the dividend. (What would be the value of `-5 % -2`? ☺)

34.3. Logical Binary Operators

Symbol	Name	Overloading Trait
`&`	Logical AND	`std::ops::BitAnd`
`|`	Logical OR	`std::ops::BitOr`
`^`	Logical XOR	`std::ops::BitXor`

When these operators, `&`, `|`, and `^`, are applied to operands of the `bool` type, they are logical AND, OR, and XOR operators, respectively. For example,

```
fn logical_demo() {
  for l in [true, false] {
    for r in [true, false] {
      print!("{l} & {r} = {b}\t", b = l & r);
    }
    println!();
  }
  for l in [true, false] {
    for r in [true, false] {
      print!("{l} | {r} = {b}\t", b = l | r);
    }
    println!();
  }
  for l in [true, false] {
    for r in [true, false] {
      print!("{l} ^ {r} = {b}\t", b = l ^ r);
    }
    println!();
  }
}
```

Here's a sample output:

```
true & true = true      true & false = false
false & true = false    false & false = false
true | true = true      true | false = true
false | true = true     false | false = false
true ^ true = false     true ^ false = true
false ^ true = true     false ^ false = false
```

34.4. Lazy Boolean Operators

Boolean expressions with binary operators, `&&` (AND) and `||` (OR), applied to operands of the `bool` type, are *lazy* expressions. That is, `&&` only evaluates its rhs operand when the lhs operand evaluates to `true`, Otherwise, the expression "short circuits" to `false`. Likewise, `||` only evaluates its rhs operand when the lhs operand evaluates to `false`, Otherwise, the expression short circuits to `true`.

For example,

```
fn lazy_boolean_operators() {
  for l in [true, false] {
    for r in [true, false] {
      let b = {
        print!("(l)"); l
      } && {
        print!("(r)"); r
      };
      print!("\t");
      print!("{l} && {r} = {b}\t");
    }
    println!();
  }
```

```
    for l in [true, false] {
      for r in [true, false] {
        let b = { print!("(l)"); l} || { print!("(r)"); r };
        print!("\t");
        print!("{l} || {r} = {b}\t");
      }
      println!();
    }
}
```

Here's a sample output. The prefix `(l)`, as opposed to `(l)(r)`, indicates that only the lhs operand has been evaluated. Notice that, in all such cases, the value of the lhs operand is the same as the value of the overall Boolean expression. This is what we refer to as the *short circuiting*.

```
(l)(r)  true && true = true     (l)(r)  true && false = false
(l)     false && true = false   (l)     false && false = false
(l)     true || true = true     (l)     true || false = true
(l)(r)  false || true = true    (l)(r)  false || false = false
```

34.5. Bitwise Operators

Symbol	Name	Overloading Trait
`&`	Bitwise AND	`std::ops::BitAnd`
`|`	Bitwise OR	`std::ops::BitOr`
`^`	Bitwise XOR	`std::ops::BitXor`
`<<`	Left Shift	`std::ops::Shl`
`>>`	Right Shift	`std::ops::Shr`

For instance,

```
fn bitwise_shift_operators() {
  let u1 = 0b1000101u8;
  println!(
    "u1 -> (u1 << 1): {0:b} ({0}) -> {1:b} ({1})",
    u1, (u1 << 1)
  );
  println!(
    "u1 -> (u1 >> 2): {0:b} ({0}) -> {1:b} ({1})",
    u1, (u1 >> 2)                          ①
  );
  let i1 = 0b1000101i8;
  println!(
    "i1 -> (i1 << 1): {0:b} ({0}) -> {1:b} ({1})",
    i1, (i1 << 1)
  );
  println!(
    "i1 -> (i1 >> 2): {0:b} ({0}) -> {1:b} ({1})",
    i1, (i1 >> 2)                          ②
  );
  let i1 = -0b1000101i8;                   ③
  println!(
    "i1 -> (i1 << 1): {0:b} ({0}) -> {1:b} ({1})",
    i1, (i1 << 1)
  );
  println!(
    "i1 -> (i1 >> 2): {0:b} ({0}) -> {1:b} ({1})",
    i1, (i1 >> 2)                          ④
  );
}
```

① The right shift operator applied to an unsigned integer extends `0`s in the leftmost bits. In this example, `(u1 >> 2)` is `00010001`.

② The right shift operator applied to a positive signed integer extends `0`s in the leftmost bits. In this example, `(i1 >> 2)` is `00010001`.

③ In Rust, signed integers are represented using *two's complement*. For example, `69` is represented as `01000101` whereas `-69` is represented as `10111011`.

④ The right shift operator applied to a negative signed integer extends `1`s in the leftmost bits. In this example, `(i1 >> 2)` is `11101110`.

Here's a complete sample output.

```
u1 -> (u1 << 1): 1000101 (69) -> 10001010 (138)
u1 -> (u1 >> 2): 1000101 (69) -> 10001 (17)
i1 -> (i1 << 1): 1000101 (69) -> 10001010 (-118)
i1 -> (i1 >> 2): 1000101 (69) -> 10001 (17)
i1 -> (i1 << 1): 10111011 (-69) -> 1110110 (118)
i1 -> (i1 >> 2): 10111011 (-69) -> 11101110 (-18)
```

34.6. The Assignment Operator (=)

An assignment expression has the form, `EXP1 = EXP2`, and it copies or moves the value of `EXP2` (a value expression) to `EXP1` (a mutable place expression), depending on their type. The assignment expression always returns chapter-primitives-unit, `()`, and they are more often than not used as statements.

For instance,

```
fn assignment_operator() {
  let mut a = 10;
  let b = 1;
  a = b + 3;                                ①
  println!("{a}");
```

```
    let mut _x = 5; let _y;
    let z = _y = (_x = 30);                ②
    println!("{z:?}");
}
```

① The rhs of this assignment is evaluated first, which results in `4`. Then, this number `4` is copied to the lhs variable `a` (since `i32` is a Copy type). As can be seen from the trailing `;`, it is an expression statement.

② Although legal, it is not very useful. The assignment expression, e.g., `_x = 30`, always returns the unit value `()`. Hence, `z` will end up being `()` as well.

34.7. Compound Assignment Operators

Compound assignment expressions combine arithmetic and logical/bitwise binary operators with assignment expressions.

Symbol	Name	Overloading Trait
`+=`	Addition Assignment	`std::ops::AddAssign`
`-=`	Subtraction Assignment	`std::ops::SubAssign`
`*=`	Multiplication Assignment	`std::ops::MulAssign`
`/=`	Division Assignment	`std::ops::DivAssign`
`%=`	Remainder Assignment	`std::ops::RemAssign`
`&=`	Bitwise AND Assignment	`std::ops::BitAndAssign`
`|=`	Bitwise OR Assignment	`std::ops::BitOrAssign`
`^=`	Bitwise XOR Assignment	`std::ops::BitXorAssign`
`<<=`	Left Shift Assignment	`std::ops::ShlAssign`
`>>=`	Right Shift Assignment	`std::ops::ShrAssign`

34.8. Comparison Operators

Equality operators, `==` and `!=`, can be overloaded by implementing the `std::cmp::PartialEq`. Likewise, other comparison operators, `<`, `>`, `<=`, and `>=`, can be overloaded by implementing `std::cmp::PartialOrd`.

Symbol	Name	Overloading Method
`==`	Equal	`std::cmp::PartialEq::eq`
`!=`	Not equal	`std::cmp::PartialEq::ne`
`>`	Greater than	`std::cmp::PartialOrd::gt`
`<`	Less than	`std::cmp::PartialOrd::lt`
`>=`	Greater than or equal to	`std::cmp::PartialOrd::ge`
`<=`	Less than or equal to	`std::cmp::PartialOrd::le`

Rust implements these traits for all primitive types, among others. One thing to note is that, unlike the arithmetic and logical operators, comparison operators use *implicit* immutable references of their operands. For example,

```
fn comparison_demo() {
  let x = String::from("Hello ");
  let y = String::from("World!");
  if x == y {                              ①
    panic!("Hello Huh?");
  }
  if PartialEq::eq(&x, &y) {               ②
    panic!("World Wot?");
  }
  println!("{x}{y}");                      ③
}
```

① `x == y` is merely a syntactic shorthand for ...

② ... `PartialEq::eq(&x, &y)`. Note that both arguments are taken through shared borrows.

③ That means that they did not move into the `eq` function.

`PartialEq` and `PartialOrd` are supertraits of `Eq` and `Ord`, respectively. While the "partial" traits can be used to compare values across two different types, `Eq` and `Ord` traits are used to *characterize* a type. An `Eq` type `T` has to satisfy an implicit requirement that `a == a` for all values `a` of `T`. Likewise, `Ord` has to support "total ordering". That is, the values of an `Ord` type should be *sortable*. On the other hand, `PartialOrd` only requires that two values should be *comparable*.

34.9. Borrow Operators

The shared borrow (`&` and `&&`) and mutable borrow (`& mut` and `&& mut`) operators are unary prefix operators used for taking references of the given operand. They cannot be overloaded. When a borrow operator is applied to a place expression, it produces an (shared or mutable) reference to the location that the value of the operand refers to. If the `&` or `&mut` operator is applied to a value expression, then a temporary memory location is created. For instance,

```
fn shared_borrow_demo() {
  let x = 101;
  let y = &x;                                    ①
  let z = &&x;                                   ②
  assert_eq!(y, &x);
  assert_eq!(z, &y);
}
```

① The type of `y` is `&i32` since `x` is of the type `i32`.

② The type of `z` is `&&i32`. Note that `&&x` is the same as `&(&x)` or `& &x`. However, `&&` is a single lexical token (because it is also used as a lazy logical AND operator), and, for example, `& & &` and `&& &` and `& &&` are all equivalent when they are used for borrows.

Likewise,

```
fn mutable_borrow_demo() {
  let (mut x1, mut x2, x3, mut x4) = (201, 301, 401, 501);
  {                                          ①
    let y1 = &mut x1;                        ②
    let z2 = &&mut x2;                       ③
    let z3 = &mut &x3;                       ④
    let z4 = &mut &mut x4;

    // println!("{x1},{x2},{x3},{x4}");     ⑤
    println!("{y1}, {z2}, {z3}, {z4}");
  }                                          ⑥
  println!("{x1}, {x2}, {x3}, {x4}");        ⑦
}
```

① A block expression creates a new scope.

② The variable `y1` mutably borrows `x1`. Note that `y1` itself is not mutable.

③ `&&mut x2` is the same as `& (&mut x2)`.

④ Note that `x3` is immutably borrowed here.

⑤ This statement will fail because `x1`, `x2`, and `x4` have been mutably borrowed.

⑥ The end of the block expression statement. Its value `()` is discarded.

⑦ This works now.

Chapter 35. Operator Overloading

Rust supports some limited form of operator overloading, using predefined `ops` and `cmp` traits, as we cover in the previous chapter. For a quick reference, we list here all traits that can be used for the purposes of operator overloading.

Arithmetic and logical operators

Trait	Op	Name	Required Methods
`Neg`	`-`	Unary negation	`neg(self) -> Output`
`Add`	`+`	Addition	`add(self, rhs: Rhs) -> Output`
`AddAssign`	`+=`	Addition assignment	`add_assign(&mut self, rhs: Rhs)`
`Sub`	`-`	Subtraction	`sub(self, rhs: Rhs) -> Output`
`SubAssign`	`-=`	Subtraction assignment	`sub_assign(&mut self, rhs: Rhs)`
`Mul`	`*`	Multiplication	`mul(self, rhs: Rhs) -> Output`
`MulAssign`	`*=`	Multiplication assignment	`mul_assign(&mut self, rhs: Rhs)`
`Div`	`/`	Division	`div(self, rhs: Rhs) -> Output`
`DivAssign`	`/=`	Division assignment	`div_assign(&mut self, rhs: Rhs)`
`Not`	`!`	Unary logical negation	`not(self) -> Output`
`Rem`	`%`	Remainder	`rem(self, rhs: Rhs) -> Output`

`RemAssign`	`%=`	Remainder assignment	`rem_assign(&mut self, rhs: Rhs)`
`Index`	`[]`	Indexing	`index(&self, index: Idx) -> &Output`
`IndexMut`	`[]`	Mutable indexing	`index_mut(&mut self, index: Idx) -> &mut Output`
`RangeBounds`	`..` `..=`	Range bounds	`start_bound(&self) -> Bound<&T>` `end_bound(&self) -> Bound<&T>`
`BitAnd`	`&`	Bitwise AND	`bitand(self, rhs: Rhs) -> Output`
`BitAndAssign`	`&=`	Bitwise AND assignment	`bitand_assign(&mut self, rhs: Rhs)`
`BitOr`	`|`	Bitwise OR	`bitor(self, rhs: Rhs) -> Output`
`BitOrAssign`	`|=`	Bitwise OR assignment	`bitor_assign(&mut self, rhs: Rhs)`
`BitXor`	`^`	Bitwise XOR	`bitxor(self, rhs: Rhs) -> Output`
`BitXorAssign`	`^=`	Bitwise XOR assignment	`bitxor_assign(&mut self, rhs: Rhs)`
`Shl`	`<<`	Bit left shift	`shl(self, rhs: Rhs) -> Output`
`ShlAssign`	`<<=`	Bit left shift assignment	`shl_assign(&mut self, rhs: Rhs)`
`Shr`	`>>`	Bit right shift	`shr(self, rhs: Rhs) -> Output`
`ShrAssign`	`>>=`	Bit right shift assignment	`shr_assign(&mut self, rhs: Rhs)`

One thing to note is that the operands of all of arithmetic and logical operators are evaluated in value expression context, which means that they are moved or copied depending on their types.

Comparison operators

Trait	Op	Name	(Required) Methods
`PartialEq`			`eq(&self, other: &Rhs) -> bool`
—	`==`	EQ	`eq(&self, other: &Rhs) -> bool`
—	`!=`	NE	`ne(&self, other: &Rhs) -> bool`
`PartialOrd`			`partial_cmp(&self, other: &Rhs) -> Option<Ordering>;`
—	`<`	LT	`lt(&self, other: &Rhs) -> bool`
—	`<=`	LE	`le(&self, other: &Rhs) -> bool`
—	`>`	GT	`gt(&self, other: &Rhs) -> bool`
—	`>=`	GE	`ge(&self, other: &Rhs) -> bool`

Unlike the arithmetic and logical operators, comparison operators implicitly take shared borrows of their operands, evaluating them in place expression context.

Here's a quick (and, trivial) example of operator overloading:

```
use std::{ cmp::Ordering, fmt::{Display, Formatter}, ops::Add };

#[derive(Debug, Clone, Copy)]
struct Int(i32);

impl Add for Int {                          ①
  type Output = Int;
  fn add(self, rhs: Self) -> Self::Output {
    Int(self.0 + rhs.0)
  }
}
```

```
impl PartialEq for Int {                    ②
  fn eq(&self, other: &Self) -> bool {
    self.0 == other.0
  }
}

impl PartialOrd for Int {                   ③
  fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
    self.0.partial_cmp(&other.0)
  }
}

impl Display for Int {
  fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
    write!(f, "{d}", d = self.0)
  }
}

fn overloading_demo() {
  let sum = Int(1) + Int(2);
  println!("sum = {sum}");                    ④
  assert!(Int(10) == Int(10));                ⑤
  assert!(Int(1) != Int(-1));
  assert!(Int(100) > Int(1));
  assert!(Int(1) < Int(100));
}
```

① Overloading the `+` operator by implementing `Add` for `Int`.

② Overloading the equality and inequality operators

③ Overloading other comparison operators.

④ This will output *sum = 3*.

⑤ All assertions will succeed.

Chapter 36. Iterators

The `Iterator` trait, and its related traits, in Rust wear many hats. Anybody who has done some functional programming knows that the list data structure plays the fundamental role in FP, no matter whether you use Lisp-style or ML-style languages. It's almost as if FP is all about just manipulating lists. In Rust, the `Iterator` trait is the abstraction of the list. All language-supported linear data types like arrays, slices, and vectors, among others, can be converted to an `Iterator` type in some way, e.g., by implementing the `IntoIterator` trait. We will not discuss this particular usage of iterators in this book.

Another important, and inherently related, role of the `Iterator` traits in Rust is to support iteration, through what is called the *iterator pattern*. This is becoming so ubiquitous across many different modern programming languages that it is rather impossible not to have used this kind of iteration idioms for anyone who has done any degree of programming these days. Unfortunately, however, different languages use different terminologies. For reference, Rust's `IntoIterator` (or, iterable) and `Iterator` correspond to C#'s `IEnumerable` and `IEnumerator` and Python's generator and iterator, respectively, among others.

Rust's iterators also support a particular style of APIs, often called the chaining, or combinators, etc., which are sometimes considered synonymous with "functional programming" (in the imperative programming world). Terms like *higher order functions* are also commonly used in this context. This kind of programming style is often based on *function compositions*, and hence it definitely reflects the core functional programming philosophy.

We will briefly go over some common uses of iterators in Rust in this chapter. Note that the idiomatic Rust programming style heavily makes use of the `Iterator`-based types. (The readers are encouraged to explore further, beyond this book, if you haven't had much exposure to this particular programming style, e.g., using `map`, `filter`, `foreach`, and `collect` style functions.)

Three Forms of Iteration

All iterables need to implement the `IntoIterator` trait, which defines one required method, `into_iter`. We won't go into all the details here, but as indicated, the `into_iter` method returns an `Iterator` type, which encapsulates all iteration logic.

```
pub trait IntoIterator {
  type Item;
  fn into_iter(self) -> Iterator<Item = Self::Item>;
}
```

We can implement `IntoIterator` for a container type `MyCon<T>` as follows:

```
impl<T> IntoIterator for MyCon<T> {
  fn into_iter(self) -> Iterator<Item = T> { /* ... */ }
  // ...
}
```

Alternatively, or in addition, we can also implement `IntoIterator` for `& MyTy` and/or `&mut MyTy` in the following way:

```
impl<'a, T> IntoIterator for &'a MyCon<T> {
  fn into_iter(self) -> Iterator<Item = &'a T> { /* ... */ }
  // ...
}
impl<'a, T> IntoIterator for &'a mut MyCon<T> {
  fn into_iter(self) -> Iterator<Item = &'a mut T> { /* ... */ }
  // ...
}
```

These three different impls will end up providing three different (albeit related) implementations for `into_iter`. Many standard collection types that implement `IntoIterator`, e.g., arrays and vectors, provide convenience methods for one or both of the last two variations, using the standard naming convention. That is, the `iter` method on type `MyCon<T>` is defined to be essentially a shorthand for `into_iter` on `&MyCon<T>`. Likewise, the `iter_mut` method on type `MyCon<T>` is defined to be effectively the same as `into_iter` on `&mut MyCon<T>`.

In other words,

- `MyCon<T>.into_iter()` returns an iterator that iterates over values of `T`.
- `MyCon<T>.iter()`, if defined, returns an iterator that iterates over values of `&T`.
- `MyCon<T>.iter_mut()`, if defined, returns an iterator that iterates over values of `&mut T`.

Here's a quick example using `Vec`:

```
fn vec_iteration_demo() {
  let mut v = vec![1, 2, 3];

  v.clone().into_iter()                    ①
    .for_each(|i| println!("{}", i));

  v.iter()                                 ②
    .for_each(|j| println!("{}", *j));

  v.iter_mut().for_each(|k| {              ③
    *k += 1;
    println!("{}", *k)
  });
}
```

① The type of `i` is `i32`, and we are iterating over `i32`. `Vec` is a Move type, and we use `clone` to be able to use the same vector for all three cases.

② We are iterating over `&i32`.

③ We are iterating over `&mut i32`.

Note that the above three iterations are essentially equivalent to the following three `for` loops, respectively.

```
fn vec_iteration_demo() {
  let mut v = vec![1, 2, 3];

  for i in v.clone() {                    ①
    println!("{}", i);
  }

  for j in &v {                           ②
    println!("{}", *j);
  }

  for k in &mut v {                       ③
    *k += 1;
    println!("{}", *k);
  }
}
```

① The type of `i` is `i32`.

② The type of `j` is `&i32`.

③ The type of `k` is `&mut i32`.

Chapter 37. Error Handling

The `try - catch` exception framework was once considered the golden standard of error handling. In a program error situation, it bypasses the normal function return flows on the call stack, and it can be rather efficient in handling errors. Virtually all widely used programming languages support some variations of exceptions, including C++, Java, C#, Python, JavaScript, just to name a few. *Rust doesn't, however.*

Exception-based error handling has pros and cons, just like everything else in *life*. In particular, it makes error handling explicit, which is an advantage. On the other hand, it makes error handling too verbose, which is a downside.

In some newer, and arguably more modern, programming languages, they do away with exceptions. Go, for example, uses the normal call stack to return error values. By convention, Go's functions return any error values as the last return value. Otherwise, it uses the `nil` value to indicate an absence of an error. Clearly, this is an improvement over C, which, by convention, uses certain special values as a return value. (Unlike in most C-style languages, Go's functions can return multiple values.) It makes error handling explicit. And, to a certain degree, it *forces* the developers to explicitly deal with possible error situations. The downside is clearly its verbosity. It *forces* the developers to deal with errors, essentially for every function call, through the call stack unwinding.

Rust's error handling is somewhat similar to Go's, which is in turn based on C's as stated, but with a twist.

By convention, all Rust functions which can cause an error return values of either `Result` or `Option` type, or something comparable. If the function call succeeds, the return value, if any, is included in the `Ok<T>` or `Some<T>` variant of `Result` or `Option`, respectively. Otherwise, the error condition is indicated by returning an `Err<E>` or `None` variant.

Now, even at a superficial level, this is a big improvement over the error handling mechanism of Go, or C. Rust's error handling is more structured, and it is possibly much more type-safe. And, unlike the exception framework, which usually stands out like a sore thumb in normal programs, it is better "integrated" into the *natural* programming styles.

37.1. Panic and Terminate

Errors are rarely black and white. There are fifty shades and one hundred nuances to program errors. However, for practical reasons, we tend to simplify their handling. In a potential error situation at run time, one of the easiest, and sometimes the best, solutions is to just to terminate the program. In certain cases, the errors can be so severe that the program may not be able to continue in any meaningful way. Sometimes, attempting to do so may be more harmful. Sometimes, there may not be a possibility of any reasonable recovery from a totally unexpected situation. In certain cases, and in fact more often than not, any elaborate error handling or recovery may not be necessary in practice. And so forth. In such cases, we can just terminate the program, e.g., by calling the `std::process::exit` function.

37.1.1. Exit

A Rust program normally ends when it runs out of the statements to execute. You can explicitly call `exit` from the `std::process` crate, for example, to terminate a program before the program's normal termination and/or to return a particular return code to the operating system, etc.

```
fn process_exit_demo() {
  println!("Hello and Bye! :)");
  std::process::exit(0);
}
```

37.1.2. The `Termination` trait

As we indicated in the beginning of the book, Rust's `main` function can return `Result<(), E>`. In fact, although it is not commonly used outside the system libraries, `main` can also return a value of `Result<T: Termination, E: Debug>`. `Termination` is a trait that was specifically introduced to support arbitrary return types in the `main` function. This trait was stabilized in Rust 1.61, and it is automatically implemented for the `Result<T, E>` type. More specifically,

```
impl<T: Termination, E: Debug> Termination for Result<T, E>;
```

Here's a simple example of the `main` function using a `Termination` type.

```
use std::{
  process::{ExitCode, Termination},
  time::{Duration, SystemTime, UNIX_EPOCH},
};

#[derive(Debug)]
struct Outcome(i32);

#[derive(Debug)]
struct Fault;

impl Termination for Outcome {           ①
  fn report(self) -> ExitCode {          ②
    if self.0 == 0 {
      ExitCode::SUCCESS
    } else {
      ExitCode::FAILURE
    }
  }
}
```

```
fn main() -> Result<Outcome, Fault> {    ③
  println!("Hello, Rust the Crab!");

  if let Ok(true) =
SystemTime::now().duration_since(UNIX_EPOCH).and_then(|d| {
    if d.as_millis() % 2 == 0 { Ok(true) } else { Ok(false) }
  }) {                                   ④
    Ok(Outcome(0))                       ⑤
  } else {
    Err(Fault)
  }
}
```

① We use the `Outcome` tuple struct type as the success return type in `Result`, and hence it needs to implement the `Termination` trait.

② Note that the `Termination` trait defines one required method, `fn report(self) -> ExitCode`.

③ A special `main` function signature.

④ *Pay no attention to that man behind the curtain.* ☺

⑤ Half of the time, it returns `Ok(Outcome)`, and for the rest half, it returns `Err(Fault)`, just for illustration.

37.1.3. Panic

Rust provides a number of macros to make it easy to terminate a program in an unexpected situation. One of the most commonly used macros is `panic!`, which does not require any further explanation. The `panic!` macro is described earlier in the book,

37.1.4. Assertions

One of the most underutilized methods in error handling is runtime assertions. They can be very effective in preventing the programs from continuing that are possibly in an inconsistent, and unsafe, state. Those programs, if they keep running, could potentially wreak havoc, e.g., by compromising the integrity of data, etc. When a certain invariant condition is not satisfied, we can cause a panic by using the `assert` macros in Rust. For example,

```
fn assert_demo() {
  let a = 1 + 2;
  assert_eq!(a, 3);                          ①

  let x = 10;
  let b = if 100 > x { true } else { false };
  assert!(b);                                ②
}
```

① If this assertion fails, then obviously something is *very* wrong, and we should not continue. ☺

② Ditto.

37.2. Results and Options

In Rust, there are two ways to program: The hard way and the *harder* way. ☺ Generally speaking, if you write programs using Rust's basic constructs, the programs tend to get rather verbose and lengthy. It sometimes gets tedious to write even a simple program. (This is the "harder" way.)

To compensate this, Rust provides an *extensive* set of APIs for virtually all types, including even primitive types. Not just Rust. Most, if not all, community libraries follow this pattern. By making use of the convenience APIs that are provided, one

can write a very "ergonomic", economical, and efficient code in Rust. There is clearly a learning curve, and it is still "hard" since it takes time and effort to get used to the APIs, but ultimately it's the *Rust way*.

As stated, Rust does not include any kind of formal error handling framework, other than providing the standard library type `Result<T, E>`, which is to be used as a return type from a function that can possibly fail. Likewise, by convention, the `Option<T>` enum type is used as a container that may or may not include a valid value.

These two types are examples of what is generally known as the *Monad* in mathematics, and in pure functional programming (a la Haskell). Although it is well beyond the scope of this book, these monads provide ideal tools in error handling. Let's suppose that the functions that can return one of two possible values, e.g., success and failure, are chained together. One can quickly realize that there will be a combinatorial explosion. That is, for each call, the situations we need to deal with will *double*. This is, in fact, a rather common problem. For example, there is this well-known problem called the "callback hell".

The interested readers are encouraged to learn more about monads, and find out how they can remedy this combinatorial explosion problem stemming from the multiple return values from a chain of function calls, for instance. But, this knowledge is not required to program effectively in Rust. To circle back on the original point that we raised in the beginning of this section, both `Result` and `Option` come with extensive sets of functions and methods to facilitate their *ergonomic* use.

We are down to the last few sections of this book, and we will not discuss any of those APIs here, but as stated, knowing, and using, them well will make you an effective programmer in Rust. In fact, many beginning Rust programmers start with the methods like `unwrap` and `expect` (whose use is not generally recommended for production code), and it's just a matter of time before they start using more sophisticated methods.

37.3. The Unwrap Operator (?)

The question mark operator, `?`, does magic in Rust. As stated, error handling can be rather tedious, not just in Rust, but in general, across many different programming languages. The question mark operator, also known as the unwrap or try operator, helps simplify, and reduce, many error handling related boiler plate code.

This special unary postfix operator can only be used with expressions of the types `Result<T, E>` and `Option<T>` (e.g., the two basic monad types in Rust error handling), and it cannot be overloaded. In either case, it unwraps valid values, from either `Ok<T>` or `Some<T>`, if successful. Otherwise, it returns the erroneous values, either `Err<E>` or `None`, to the calling function.

37.3.1. Unwrapping `Option`

When the unwrap operator is applied to an operand of the `Option<T>` type,

- If the operand expression evaluates to `Some(x)`, then
  - It will automatically unwrap the value, and hence the overall expression will evaluate to `x`, and
- If the operand evaluates to `None`, then
  - It will cause the enclosing function or closure to immediately return with the value, `None`.

For example, if a function `o()` is defined as follows:

```
fn o() -> Option<T> { /* ... */ }
```

Then, the following two functions are more or less equivalent to each other.

```
fn use_o() -> Option<T> {
  let x = o()?;
  // Use x...
}
```

```
fn use_o() -> Option<T> {
  match o() {
    Some(x) => // use x...
    None => return None,
  }
}
```

37.3.2. Unwrapping `Result`

When the unwrap operator `?` is applied to an operand of a `Result<T, E>` type,

- If the operand expression evaluates to `Ok(x)`, then
  - It will automatically unwrap the value and hence the overall expression will evaluate to `x`, and
- If the operand evaluates to `Err(e)`, then
  - It will cause the enclosing function or closure to immediately return with the error value, `Err(From::from(e))`.

For example, if a function `r()` is defined as follows:

```
fn r() -> Result<T1, E1> { /* ... */ }
```

Then, the following two functions are more or less equivalent to each other.

```
fn use_r() -> Result<T2, E2> {
  let t1 = r()?;
  // Use t1...
  // Return Ok(t2), with t2 being of the type T2.
}
```

```
fn use_r() -> Result<T2, E2> {
  match r() {
    Ok(t1) => // use t1, and return Ok(t2)
    Err(e1) => return Err(e2), // with e2 = From::from(e1) unless E1
== E2, in which case e1 == e2.
  }
}
```

Note that, in the first form, the error type conversion, if necessary, is implicit in the unwrap operator expression.

37.4. Error Type Conversions

As we showed in the previous section, some error types used in a program may need to be *convertible* to one another, e.g., by implementing the `From` trait, in order to be able to take advantage of the elegant syntax of the unwrap operator `?`.

Here's a quick example illustrating this point.

```
#[derive(Debug)]
struct Error1(u8);                          ①

#[derive(Debug)]
struct Error2(u16);
```

① We use two simple types as error types in this example. Rust includes the `Error` trait, but it is not necessary to use this trait. At least, for now.

```
impl From<Error1> for Error2 {           ①
  fn from(value: Error1) -> Self {
    Error2(value.0 as u16)
  }
}
```

① The type `Error2` implements the `From<Error1>` trait, which means that values of type `Error1` can be converted to values of type `Error2`.

```
fn error_1() -> Result<i32, Error1> {    ①
  Err(Error1(1))
}
```

① This `error_1` function merely returns an error of type `Error1`.

```
fn error_2() -> Result<i32, Error2> {    ①
  let x = error_1()?;                    ②
  println!("{x}");                       ③
  Ok(0)
}
```

① The `error_2` function returns an error of type `Error2` in case of errors.

② `error_1()?` can potentially return a value of `i32`, in which case we assign the value to the variable `x`, or it can just return an error, in which case the error returned from `error_1()` will be converted using the `From::from` function.

③ Note that if the program execution reaches this point, then `x` will always have a valid value.

```
fn error_2() -> Result<i32, Error2> {
  Ok(error_1()?)                          ①
  // Ok(error_1()?.abs())
  // Ok(error_1()? + 2)
}
```

① Since the overall expression postfixed with the unwrap operator is also an expression, it can be used anywhere an expression is expected. Note that, in this example, the type of these expressions are `i32`. The unwrap operator expressions can also be chained.

Builtin Attributes

The `test` Attribute

Although we do not discuss Rust's unit testing support in this book, it might be a good idea to end the book with a description of the `test` attribute. One can run unit tests via *rustc --test* or *cargo test*. All functions marked with the `test` attribute, but not with the `ignore` attribute, will be run as unit tests.

The test function should be one of the following three forms:

```
#[test] fn test_1() -> () {}
#[test] fn test_2() -> ! {}
#[test]
fn test_3() -> Result<T: Termination, E: Debug> {
  todo!();
}
```

Appendix A: How to Use This Book

> Tell me and I forget. Teach me and I remember. Involve me and I learn.
>
> — Benjamin Franklin

The books in this "Mini Reference" series are written for a wide audience. It means that some readers will find this particular book "too easy" and some readers will find this book "too difficult", depending on their prior experience related to programming. That's quite all right. Different readers will get different things out of this book. At the end of the day, learning is a skill, which we all can learn to get better at. Here are some quick pointers in case you need some advice.

First of all, books like this are bound to have some errors, and some typos. We go through multiple revisions, and every time we do that there is a finite chance to introduce new errors. We know that some people have strong opinions on this, but you should get over it. Even after spending millions of dollars, a rocket launch can go wrong. All non-trivial software have some amount of bugs. If you are fixated over a few typos, which is probably less than 0.001% of the total word count in the book, then this book is not for you.

Although it's a cliche, there are two kinds of people in this world. Some see a "glass half full". Some see a "glass half empty". *This book has a lot to offer*. As a general note, we encourage the readers to view the world as "half full" rather than to focus too much on negative things. *Despite* some (small) possible errors, and formatting issues, you will get *a lot* out of this book if you have the right attitude.

There is this book called *Algorithms to Live By*, which came out several years ago, and it became an instant best seller. There are now many similar books, copycats, published since then. The book is written for "laypeople", and illustrate how computer science concepts like specific algorithms can be useful in everyday life.

Inspired by this, we have some concrete suggestions on how to best read this book. This is *one* suggestion which you can take into account while using this book. As stated, ultimately, whatever works for you is the best way for you.

Most of the readers reading this book should be familiar with some basic algorithm concepts, at least at a high level, although you may not remember all the details. When you do a graph search, there are two major ways to traverse all the nodes in a graph. One is called the "depth first search", and the other is called the "breadth first search". At the risk of oversimplifying, when you read a tutorial style book, you go through the book from beginning to end. Note that the book content is generally organized in a tree structure. There are chapters, and each chapter includes sections, and so forth. Reading a book sequentially often corresponds to the *depth first traversal*.

On the other hand, for reference-style books like this one, which are written to cover broad and wide range of topics, and which have many interdependencies among the topics, it is often best to adopt the *breadth first traversal*.

This advice should be especially useful to new-comers to the language. The core concepts of any (non-trivial) programming language are all interconnected. That's the way it is. When you read an earlier part of the book, which may depend on the concepts explained later in the book, you can either ignore the things you don't understand and move on, or you can flip through the book to go back and forth. It's up to you. One thing you don't want to do is to get stuck in one place, and be frustrated and feel resentful (toward the book).

In essence, the best way to read books like this one is through "multiple passes", again using a programming jargon. The first time, you only try to get the high-level concepts. At each iteration, you try to get more and more details. It is really up to you, and only you can tell, as to how many passes would be required to get much of what this book has to offer.

Again, *good luck!*

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

S

T

U

V

W

X

Z

About the Author

Harry Yoon has been programming for over three decades. He has used over 20 different programming languages in his academic and professional career. His experience spans broad areas from scientific programming and machine learning to enterprise software and Web and mobile app development.

He *occasionally* hangs out on social media:

- Instagram: @codeandtips [https://www.instagram.com/codeandtips/]
- TikTok: @codeandtips [https://tiktok.com/@codeandtips]
- Twitter: @codeandtips [https://twitter.com/codeandtips]
- YouTube: @codeandtips [https://www.youtube.com/@codeandtips]
- Reddit: r/codeandtips [https://www.reddit.com/r/codeandtips/]

(Upcoming) Rust Books by the Author

- Rust CLI and Text UI Programming: Learn and Build Command Line and Terminal Applications with Rust.
- Basic Algorithms in Rust: Linear Data Structures and Algorithms for Beginners and Non-Computer Science Majors
- Functional Programming with Rust: Introduction to Theories and Practices in Modern Functional Programming
- Rust Web Assembly Programming: Introduction to Web Assembly with Rust - Wasm Core, Web, JavaScript, and WASI APIs

About the Series

We are creating a number of books under the series title, *A Hitchhiker's Guide to the Modern Programming Languages*. We cover essential syntax of the 12 select languages in 100 pages or so, Go, C#, Python, Typescript, Rust, C++, Java, Julia, Javascript, Haskell, Scala, and Lua.

These are all very interesting and widely used languages that can teach you different ways of programming, and more importantly, different ways of thinking.

All Books in the Series

- Go Mini Reference [https://www.amazon.com/dp/B09V5QXTCC/]
- Modern C# Mini Reference [https://www.amazon.com/dp/B0B57PXLFC/]
- Python Mini Reference [https://www.amazon.com/dp/B0B2QJD6P8/]
- Typescript Mini Reference [https://www.amazon.com/dp/B0B54537JK/]
- Rust Mini Reference [https://www.amazon.com/dp/B09Y74PH2B/]
- C++20 Mini Reference [https://www.amazon.com/dp/B0B5YLXLB3/]
- Modern Java Mini Reference [https://www.amazon.com/dp/B0B75PCHW2/]
- Julia Mini Reference [https://www.amazon.com/dp/B0B6PZ2BCJ/]
- Javascript Mini Reference [https://www.amazon.com/dp/B0B75RZLRB/]
- Haskell Mini Reference [https://www.amazon.com/dp/B09X8PLG9P/]
- Scala 3 Mini Reference [https://www.amazon.com/dp/B0B95Y6584/]
- Lua Mini Reference [https://www.amazon.com/dp/B09V95T452/]

Community Support

We are building a website for programmers, from beginners to more experienced. It covers various coding-related topics from algorithms to machine learning, and from design patterns to cybersecurity, and more. You can also find some sample code in the GitLab repositories.

- www.codeandtips.com
- gitlab.com/codeandtips

Mailing List

Please join our mailing list, join@codingbookspress.com, to receive coding tips and other news from **Coding Books Press**, including free, or discounted, book promotions. If we find any significant errors in the book, then we will send you an updated version of the book (in PDF). Advance review copies will be made available to select members on the list before new books are published.

Request for Feedback

If you find any errors or typos, or if any part of the book is not very clear to you, or if you have any general suggestions or comments regarding the book, then please let us know. Although we cannot answer all the questions and emails, we will try our best to address the issues that are brought to our attention.

- feedback@codingbookspress.com

Please note that creating and publishing quality books takes a great deal of time and effort, and we really appreciate the readers' feedback.

Revision 1.2.3, 2023-04-28