1

Game of Life: The Basics

IN THIS CHAPTER, YOU WILL LEARN THE FOLLOWING:

- How to create a new project using Cargo
- How to use variables in Rust
- How to use basic functions in Rust, including returning values and passing parameters
- How basic control mechanisms work

In 1970, British mathematician John Horton Conway devised a game using cellular automata. In October of that year, Martin Gardner wrote about the game in his monthly column Mathematical Games in *Scientific American*. It's a game with simple rules, which can be played on paper, but honestly, it's more fun to write programs that implement the game. We're going to start the dive into Rust by writing a simple implementation of *Conway's Game of Life*. First we'll talk about the rules so that when we get to implementing it, you'll know what you are looking at.

Imagine a two-dimensional space that consists of cells on both the horizontal and vertical axes. Maybe it's just easier to think about graph paper—row upon row and column upon column of little boxes. Each of these little boxes contains, or at least has the potential to contain, a living creature—a single-celled organism living in a single cell. The game is evolutionary, meaning we cycle through one generation after another, determining whether each cell lives or dies based on the rules of the game. Speaking of those rules, they are as follows:

- If a cell is currently alive but it has fewer than two neighbors, it will die because of lack of support.
- If a cell is currently alive and has two or three neighbors, it will survive to the next generation.

- If a cell is currently alive and has more than three neighbors, it dies from overpopulation (lack of resources).
- If a cell is currently dead but has exactly three neighbors, it will come back to life.

To turn this game into code, we need to do a couple of things. First, we need a game grid where all of our little cells are going to live. Second, we need a way to populate the game grid with some living cells. An empty game board won't lead to anything good. Once we have a game board, we can run generations using these rules.

The following is the complete program that will create the game board and also run the checks for whether different cells live or die. Don't worry—you don't have to take it all in at once. We'll go through it step-by-step as we introduce you to Rust.

GAME OF LIFE: THE PROGRAM

The program in this section will create the game board for *Conway's Game of Life* and populate it with an initial generation. This portion of this program will be more than enough to get us started talking about how to begin a Rust program. However, this is not a complete program in the sense that it won't fully implement a useful *Conway's Game of Life*. It's primarily missing the output and generational functions.

```
extern crate rand;
use std::{thread, time};

fn census(_world: [[u8; 75]; 75]) -> u16
{
    let mut count = 0;

    for i in 0..74 {
        for j in 0..74 {
            if _world[i][j] == 1
            {
                count += 1;
            }
        }
    }
    count
}
fn generation(_world: [[u8; 75]; 75]) -> [[u8; 75]; 75]
{
    let mut newworld = [[0u8; 75]; 75];

    for i in 0..74 {
        for j in 0..74 {
            let mut count = 0;
            if i>0 {
                count = count + _world[i-1][j];
            }
```

```
                if i>0 && j>0 {
                    count = count + _world[i-1][j-1];
                }
                if i>0 && j<74 {
                    count = count + _world[i-1][j+1];
                }
                if i<74 && j>0 {
                    count = count + _world[i+1][j-1]
                }
                if i<74 {
                    count = count + _world[i+1][j];
                }
                if i<74 && j<74 {
                    count = count + _world[i+1][j+1];
                }
                if j>0 {
                    count = count + _world[i][j-1];
                }
                if j<74 {
                    count = count + _world[i][j+1];
                }

                newworld[i][j] = 0;

                if (count <2) && (_world[i][j] == 1) {
                    newworld[i][j] = 0;
                }
                if _world[i][j] == 1 && (count == 2 || count == 3) {
                    newworld[i][j] = 1;
                }
                if (_world[i][j] == 0) && (count == 3) {
                    newworld[i][j] = 1;
                }
            }
        }
        newworld
    }

    fn main() {
        let mut world = [[0u8; 75]; 75];
        let mut generations = 0;

        for i in 0..74 {
            for j in 0..74 {
                if rand::random() {
                    world[i][j] = 1;
                } else {
                world[i][j] = 0;
                }
            }
        }
    }
```

STARTING WITH CARGO

Although you can certainly use just the Rust compiler, `rustc`, Rust comes with a utility that can be used to create the files and directory structure necessary to build a program that could go beyond a single file if needed. To get started, we can run `cargo new life` to create everything we need initially.

What you will get is a directory named `src`, which contains a single file, `main.rs`. Initially, you will have a simple hello, world program in that file, which means there is at least one line of code you will need to delete if you want to do something interesting. The file does, though, contain the bones of a main function. If you are familiar with C programming, you are familiar with the main function. This is the entry point for your program. When the compiler runs, the resulting executable will point to the chunk of code that results from whatever is in your main function. This function is essential for your program to do anything, because the compiler will look for it in order to know where to link the entry point (which is just an address in the `.text` segment of the resulting assembly language code).

In addition to the `src` directory and the `main.rs` file, where you will be doing all your development work initially, there is a `Cargo.toml` file. This is the configuration file used by Cargo, written in Tom's Obvious, Minimal Language (TOML). It's an easy language to use, and Cargo will put almost everything you will need into it. We will eventually get into making changes to it, but what you will see initially is metadata about the resulting executable, including your name, your email address, and the version number. Everything is in text, as you can see here in what was created when I ran `cargo new life`:

```
[package]
name = "life"
version = "0.1.0"
authors = ["Ric Messier <kilroy@mydomain.com>"]

[dependencies]
```

You will get something that looks slightly different, of course, since you have neither my name nor my email address. The version will be 0.1.0 initially, and if you actually use *life* as the name of your program, you will get that configured in your `Cargo.toml` file. Cargo takes care of all that for you.

> ***NOTE*** *Don't get too fancy with your naming. This is going to be the name given to the executable that results from building your program. If you get too fancy and try using something like camel case, Cargo will complain. It expects simple naming. If you are unfamiliar, camel case is mixing upper and lowercase letters, usually with the uppercase letter coming in the middle of the word, as in* `myProgram`.

Cargo is also used to build your project. To build your executable, you just run `cargo build`. By default, Cargo will build a debug version, which will be dropped into the target/debug folder. If you want a release version rather than a debug version, you have to run `cargo build --release`. This will place your executable into the target/release directory. You can run your program from there, should the build succeed. You will get more than the executable in the target directories.

Here, you can see the contents of the debug directory from a build of the Life program:

DEBUG DIRECTORY LISTING

```
kilroy@milobloom:~/Documents/rust/life/target$ cd debug
kilroy@milobloom:~/Documents/rust/life/target/debug$ ls
build        examples      life        life.dSYM
deps         incremental   life.d      native
```

The file named `life` is the executable, and the debug symbols are in the file named `life.dSYM`. This is useful in the case where you need to perform debugging using a debugger that will make use of these symbols to keep track of where in the program it is so that it can show not only the assembly language representation of the program but also the source code, which is likely far more meaningful than assembly language to most people. For our purposes, you won't need the debug symbols, unless you really want them, since I'll have done all the debugging to ensure all the code compiles and runs on the version of Rust that is current as of this writing.

PUTTING THE PIECES TOGETHER

Once you have created your new project using Cargo, you can start adding code, typically to the `main.rs` file. Everything we're doing going forward will be in the `main.rs` file unless specified otherwise. We'll go through the program a little at a time to explain it all. We're going to try to keep the bouncing around the program to a minimum, though there will be a little of that. To begin with, though, we'll start at the top of the file.

Bringing In External Functionality

No matter what kind of program you're writing, you'll likely need to bring in functionality from outside your own code. There are a couple of different ways to do that. We can talk about both of them here since both are in use in our Life program. The relevant code fragment is shown here. You will notice that a few different things are going on here that may be slightly different from what you're used to in other programming languages.

```
extern crate rand;
use std::{thread, time};
```

Rust uses libraries called *crates* to store external, reusable functionality. No one should be reinventing the wheel every time they write a program, so you'll probably use a lot of crates as you go. The difference between the previous two lines is the first one refers to an external crate, meaning it's a package available outside of the standard library. The library we're using here is one that will give us the ability to generate random numbers. When it comes to populating the game board on the initial world creation, you can (1) do it by hand as the programmer, (2) allow a user to do it by hand using some configuration, or (3) generate the world using random values. We'll choose the third approach on this initial pass through the world, so we need to have functions that can generate random values for us. This is not functionality included in the standard library. The `extern` keyword indicates the compiler needs to be looking elsewhere for the library.

Speaking of the standard library, the second line in the previous code brings in functionality from the standard library. We are pulling in two separate modules from the standard library, but rather than taking up two lines to do it, we're compressing it onto a single line. The `{}` you see are borrowed from Unix and they are used to mean "insert each of the values in the set contained within these brackets to complete the expression." What we are doing is just a shorthand notation that will achieve the same results as if we'd written the following two lines. This works only if you are importing functionality from the same location.

```
use std::thread;
use std::time;
```

You may be familiar with the idea of importing functionality. In a language like C, you'd include the same functionality from the C libraries using these lines:

```
#include <threads.h>
#include <time.h>
```

Other languages have the same concept of importing external functionality. In Objective-C, for instance, you can use `@import`. In Swift, you would just use `import`. C++ inherits the same include statements that C uses. One of the differences between C/C++ and other languages is that C/C++ makes use of a preprocessor that replaces directives like `#include` with actual C code. The compiler never sees the `#include` statement because it gets replaced by the preprocessor before the compiler gets to it. C++ is really just another preprocessor. All C++ code gets converted to actual C, which is then passed into the C compiler. Not all languages have a preprocessor. Rust makes use of these import statements in conjunction with Cargo, which acts less as a preprocessor and more as a coordinator.

As mentioned, the `extern` keyword indicates we are using an external library. We rely on Cargo to make sure that the library is in place and built so that when it comes time to compile the program, all external references can be successfully resolved. This means we need to add a line to our `Cargo.toml` file. In the `[dependencies]` section, we need to tell Cargo that we're going to require a library. As you can see in the following code, we provide the name of the library as well as the version number necessary for our program to work. This last part can be replaced with an `*` to indicate that any version will work, but you may need a specific version, since different versions will sometimes have different functionality, as well as different signatures.

```
[dependencies]
rand = "0.7.2"
```

The signature is important, because it identifies the parameters a function expects to receive as well as the value or values the function will return. If the program we're writing doesn't make use of the function in the same way as it is specified in the library version being used, the compilation will fail. As a result, it's important to know which version of the library you're using to ensure that you're using functions in the same way as they're specified in that one version.

Namespaces

This brings up the idea of namespaces, though this is not what Rust calls them. It's a useful concept to talk about, though, even if it's not terminology that Rust uses. Namespaces are common things, and they are especially used in object-oriented languages like C# or C++. They are also used in containers,

which are ways of virtualizing applications. A namespace really is just a container. It's a way of placing a lot of related things into the same place in order to make referring to those things consistent. This is why bringing up namespaces here makes some sense. Earlier, we brought in functionality from modules. You can think of all the properties and functions within those modules as belonging to the same namespace, by which I mean that in order to refer to them, you'd use the same naming structure.

One of the guidelines for writing programs is that we try to name functions and variables in ways that will make sense to people who are writing programs using the functions and variables we've created. In doing that, unfortunately, many modules or libraries will have functions or properties that use the same names. We need a way to differentiate one from another.

Consider your house. You have a number of rooms in your house. Each room has at least one light switch. If I were to tell you to turn off the light switch, how would you know which light switch to turn off? The room provides the context, or the namespace, that will help us make sense of the request. Then I can say turn off the light switch in the living room, and you'll know exactly what to do. You've already seen something along these lines in the previous code. When we brought in functionality from the standard library, we used `std::thread`, as one example. That expression provides us the namespace, essentially, to differentiate a thread out of the standard library from a thread from a different library.

We can take this example a little bit further, which will also move us ahead a bit. Using the Rust syntax, I can tell Rust to turn off the light in the living room using something like `livingroom::switch.off()`. This gives me the context, or namespace, up front. I'm using `livingroom` as the module I want to use functionality from. I'm going to switch out of the `livingroom` module, and then I'll call `off()` as a function or method on that switch object.

We have to keep using the namespace to refer to any object we use from modules we're making use of (`livingroom::`) in order to ensure we're clear about exactly which object we'll be using. That way, the compiler has nothing to guess about, and perhaps as importantly, when it comes to any other programmer reading what we've written, it's clear. This explicitness is something we'll keep coming back to when using Rust. Everything is explicit and is required to be explicit so that there are no guessing games or misunderstandings when it comes to what we've written versus what the compiler is generating for us. It's, frankly, one of Rust's charms.

GENERATING THE GAME GRID

With our functionality imported, we can get started writing the program. As mentioned, this is mostly going to be a linear process from the standpoint of reading the source code. As best as we can, we'll go from the top to the bottom of the source code. The one deviation is going to be that we'll start with the main function, or the entry point to the program.

One reason for putting the main function at the bottom of the source code, even though it's really the start of the program, is a holdover from C. In the C programming language, as well as with many other programming languages, you can't use something that hasn't been defined. When you write your main function, you're going to be calling other functions. If you try to call them before they've been defined or implemented (which is a definition, by definition), you'll get a compiler error because

the compiler doesn't know what it is in order to match it up against how you're using it. This is that signature thing again. If I define a function as taking two integers but you try calling it with an array of characters, that's not going to work well. The compiler should flag that, but it can't if it doesn't know what it's supposed to look like before it's used.

In Rust, you can put the main function at the top of your source code since it will hold off on passing judgment on whether you've called a function correctly until it actually sees the definition. As an exercise, take the source code from this chapter and move the main function starting with `fn main` all the way to the last } and put it at the top of the file, right under where we pull in the modules we're going to be using. When you build, it will build successfully. As we go forward and you start writing your own Rust programs, you can feel free to put the main function, or any function for that matter, wherever in the file you want. The compiler won't error on you simply because of that.

DISSECTING MAIN

We're going to the bottom of the file and looking at the main function, but in pieces because it's a fairly long function. There are also some critical components of the main function here, so we'll try to keep it slow and manageable so you'll easily understand not only the syntax of the language but also the important features that separate Rust from other languages. Where it's helpful, we'll take a look at how Rust compares with other common languages that you may be familiar with.

Defining Functions

Functions are a common feature of most languages today, though you may hear the term *method* used sometimes to describe the same sort of feature. A function is a way of putting code and data together in a smallish block. When we create functions, we create the ability to reuse a set of code over and over without having to rewrite the same code every time we want to use it. Typically, functions take parameters and may also return values. This means we can pass data into the function to operate on, and then the function can return the result of any work done to the calling function.

Rust requires the use of functions, which differs from some languages you may be familiar with. Python, for instance, does not require that you use any function. If you want, you can write a Python script without using any functions at all. Other scripting languages, similarly, don't require the use of any function. Of course, Rust isn't a scripting language like Python is. Unlike Python, Perl, or other scripting languages, Rust uses a compiler to generate an executable that is used when a user wants to run the program. Even if you do use functions when you're writing a Python program, you don't have to create a main function, which explicitly tells the interpreter (the compiler in the case of Rust) where to start the program execution.

```
fn main() {
```

Here, you can see the definition of the main function in Rust. This is a basic definition. We use `fn` to indicate that what is coming is a function. This is similar to a language like Python, which uses `def` to indicate the definition of a function. Swift uses `func` to indicate what is coming is a function. Even though these languages are said to be C-like—because some of the syntax and control structures are

similar between C and languages like Swift, Python, and Rust—the function definition in C is different. A C main function is defined as follows:

```
int main (int argc, char **argv) {
```

Rather than indicating up front that what we have is a function, we start with the variable type that the function will return at the end. In C, you have to specify some datatype to return, even if it's `void`, which is no datatype, indicating there is no return value. Languages like Rust may never return a value and if there's no value being returned, there's no indication of a value being returned, as you can see in the declaration of the main function earlier. We can absolutely return values from any function we want, and you'll see how that works later on in this chapter when we take a look at some other functions in our program. Similarly, the C declaration of the main function includes command-line parameters being passed into the main function. This is not required, just as it's not required in our Rust program. When it's not required, we simply don't include it.

Functions, as much as anything, are scope definitions. When we have data in a function, the data stops being available once we pass outside of the function. This means we need a way to indicate where the function starts and where it stops. Python likes the idea of using white space to clearly define scope. It's part of the language definition. There are no begin/end blocks with Python. You simply have to pay attention to the level of indentation. Personally, I'm not a fan of using white space as part of the syntax or definition of the language. Fortunately, Rust again follows C here. C uses curly braces (or brackets) to indicate the beginning and ending of any block of code. We start a function with a `{` and close it with a `}`. This may be harder to parse visually than the white space used in Python, but you can just use good indentation practices to give you that visual parsing ability without it being forced on you.

At this point, we have a declaration of our main function as well as the start of the code block. We can move right into the rest of the function.

Defining Variables

Some languages are really picky about where you define variables. It's usually a good practice to define all your variables at the top of a function, but it's not required by the language definition or the compiler. It makes it easier to understand what is going on if you know exactly where to look for the different elements of a function. Defining variables mid-function can make it harder to debug or read the program later on because you might miss the declaration to know what datatype is being used when you read through complex or longer functions. Using this guidance, the declarations (with one exception, which we'll get to later) are done at the top of the function.

```
    let mut world = [[0u8; 75]; 75];
    let mut generations = 0;
```

We're defining two variables in our main function. One of these is the game grid, which is a multidimensional array. Before we get to that aspect of the definition, we should address the rest of it, starting from the left side. First, we declare a variable using the keyword `let`. If we want, we can do a simple declaration of a variable by saying something like `let count = 0;`. This indicates that we have a variable named `count` that we have set to an initial value of 0. Rust will infer the datatype because we haven't specified it. Since it's defined, we can go on our merry way using the variable `count`.

> ***NOTE*** *When it comes to naming variables, you can use letters, digits, or the underscore character. You can't use special characters in the name of a variable. There are some conventions when it comes to naming, which the compiler will help you with, making suggestions when you aren't following the naming conventions. The starting character in a variable name has to be either a letter or an underscore. It's also worth noting that variable names are case sensitive. Camel case is commonly used in Java and other languages, but it is used in Rust only in specific situations, which you'll learn about in later chapters.*

This is a bit of a gotcha, however, which brings us to the second keyword in our declaration lines. It's important to note that Rust uses what the developers call *immutable variables* by default. You can quibble, like me, with the term immutable variable since variable, by definition, means changing and immutable means not changing. The term immutable variable means something that's going to change but that isn't going to change. Essentially, if you have an immutable variable, you have a constant, because it won't change. From a language and compilation perspective, an immutable variable is different from a constant.

Linguistic quibbles aside, this is an important aspect to the language. Because it's such a subtle thing, you'll see it come up a lot as we talk about different variables and how they're used throughout the rest of this book. A constant, from the perspective of the language and the compiler, is essentially an alias. Compilers, like those commonly used in the C language, will go through and simply replace the term for what it refers to. For instance, again using C as an easy way to demonstrate this concept, here's how we would declare a constant in a C program:

```
#define MYCONST 42
```

This indicates that we have a term, `MYCONST`, that refers to the value 42. The C preprocessor will run through all the code where this definition applies and replace anywhere it finds `MYCONST` with the value 42. The only purpose `MYCONST` serves is to make it easier to change the value `MYCONST` at any point and have that change be made across an entire program. It also provides some self-documentation if you give it a meaningful name. If you were to use `MAX_X`, for instance, you'd know that it would be the maximum value along the x-axis on a graph, potentially. This is more useful than just a raw number.

A variable that can't be changed is different. For a start, you can't set a constant to the result of a function call, because it's not known at compile time. You can set a variable to the return value from a function call, though once the value is set it can't be changed. A variable that can't be changed is also protected from modification, so you can always be sure that the value you expect to be there will be there—or at least that the value that was set at one point hasn't been corrupted. This helps with any concurrent programming since you can use a variable without fear of it being modified mid-use by another thread.

> ***NOTE*** *There is a concept in programming called a* pure function. *A pure function is one that will return the same value every time the function is called, given the same set of inputs. Additionally, a pure function causes no side effects, meaning there is no alteration of variables or arguments. Using non-mutable variables can help with the implementation of pure functions because we can protect against side effects. A pure function, because it has predictable outcomes, can be "proved," meaning we can test against the output to be sure the function is working as expected. This testing repeatability using automation can result in more robust programs.*

To make a change to a variable during program execution, we have to specify that it is mutable, meaning we expect it to change. We set a mutable variable with the `mut` keyword. Both of the variables being declared in the main function in this program are mutable. One of these variables is the game grid. This has to be mutable because we're going to keep changing all the values as we go through one generation to another. Cells are going to die and be born, so we need to change values in each of the positions of the array. The other variable is the generation count. This is not an absolutely necessary value other than it's interesting to keep track of what generation number we're in as the game iterates through generation after generation. Since we're going to increment that value after each generation, it has to be mutable.

It's always worth considering whether you have to have a value that is mutable or not mutable. If you're going to set it once and not touch it again, you don't need to have it mutable. You can protect your program by just leaving it immutable. This is where the compiler is helpful. If you set a value once on a variable you have indicated is mutable and then don't change it, the compiler will prompt you that it should probably be left immutable. Similarly, if you have a value that simply should not change at all, leave it immutable and if any part of the program tries to change it, the compiler will complain about it.

This compiler error can help you track down bugs faster since your compile will simply fail, and you'll have to decide whether the variable can be mutable or if the change in value should simply never have happened to begin with. If the compiler hadn't errored on you, you would've had a bug in your program later on when a value you didn't expect to change got changed. It's this explicit programming that can lead to more robust programs—if you want to change a value, you have to think about it and then indicate that the value is going to change at some point.

Datatypes

The game grid itself is where we get explicit about the type of data that is going to be used. As discussed earlier, Rust may infer the datatype based on the value that's being put into a variable, but we can also be explicit about it, and you can see this in the declaration of the `world` variable. In addition to being an array, which we'll deal with shortly, you can see that the `world` identifier has

an interesting notation where the datatype could or should be. What you'll see there is `0u8`. Rust is a strongly typed language, and you can't just move from one type to another willy-nilly.

The `0u8` is saying that we'll populate this variable with a 0 value but that the 0 value is going to be an unsigned 8-bit integer. This allows us to initialize the value at the same time we tell Rust (the compiler in this case) the datatype to expect. This means we never expect to get a value larger than 255 in this field. Because it's unsigned, we aren't ever going to have to accommodate a signed bit, so we can take values from 0 to 255 in a u8 datatype. As you might expect, if we can support unsigned, we can support signed as well. A signed 8-bit integer would be declared by i8.

This is another area where Rust lets you be as explicit as you want to be. Depending on your memory requirements, you can pick whatever size you want for your integer values. You can use 8-, 16-, 32-, 64-, or 128-bit values, both signed and unsigned. This means that you can declare variables to be i8, i16, i32, i64, i128, u8, u16, u32, u64, or u128. You can also specify the size of your floating-point values, though you get the choice of f32 and f64 only. The default floating-point size is 64 bits because it has no performance penalty on modern processors but has considerably more precision.

We are not limited to just numbers, though. We can also create *char* values. A char in Rust is a 4-byte value, which allows for support of Unicode values as well as accents and emoji characters. It's perfectly legal in Rust to do the following, assuming your editor allows you to enter this character:

```
    let emo_char = '☺';
```

Another common datatype is the Boolean value. A Boolean value, used for logic operations, will evaluate to true or false. If we wanted to create a Boolean value and use explicit type annotation, we'd use the following statement:

```
    let yes_no: bool = true;
```

This statement lets us declare the datatype while setting the value at the same time. Someone who is accustomed to other languages may find it difficult to get used to using the *`variable: datatype`* notation ahead of an equal sign to set the value. It can also be challenging to read initially if you're accustomed to languages like C, C++, Java, C#, and others where you indicate the datatype on the left-hand side, ahead of the variable name. In this case, Rust uses the keyword `let` to indicate there's a variable here, and so it needs another way of declaring variables. It might be even more awkward to use `let` *`datatype variable`* `=` *`value`*. Either way, we don't get a vote here, so you'll have to accept `let` *`datatype variable`* `=` *`value`* as the way you declare and set initial values on variables.

It's worth noting that, just because you don't want this to turn into a gotcha, the variables we created are immutable. The value can't be changed. This also raises the importance of naming. Using a variable name `yes_no` on a Boolean value that can't change after it's been set to true isn't really a good way of naming it. It is always going to, effectively, be yes and will never be no. So, two lessons from our earlier declarations. Always think about whether you are going to make a variable mutable and then make sure you are giving the variable a meaningful name so that you can read and understand it later. Or, perhaps, someone else can read and understand it.

Arrays

One of the variables we are going to work with is an array. More specifically, it's a multidimensional array. An array isn't a datatype itself. It's a primitive data structure. There are better ways of handling

data that is tightly related and you want to be able to address it directly, as in either walking through the entire data stream or just going straight to a particular value. The problem is that none of the other ways of handling this data structure can handle multiple dimensions. Imagine a single-dimension array, or even better, just take a look at Figure 1.1, which shows a single-dimension array. This would be a chunk of contiguous memory where you would store a number of values.

FIGURE 1.1: Single-dimension array

One important aspect to consider here is that when we are working with arrays, all the values will have the same datatype. In our case, we have a collection of unsigned 8-bit values. In reality, we're only going to be using two values. We could use an array of Boolean values, true or false, but using unsigned integers means we can do arithmetic directly with the values we have. This gives us a couple of ways of keeping track of how many neighbors our cells have—we just add up all the values or we check to see whether there is a value and then increment. For our "world," we are going to be using a multidimensional array, which in practice is going to look like Figure 1.2, though in reality it will just be a contiguous section of memory, just like a single-dimension array.

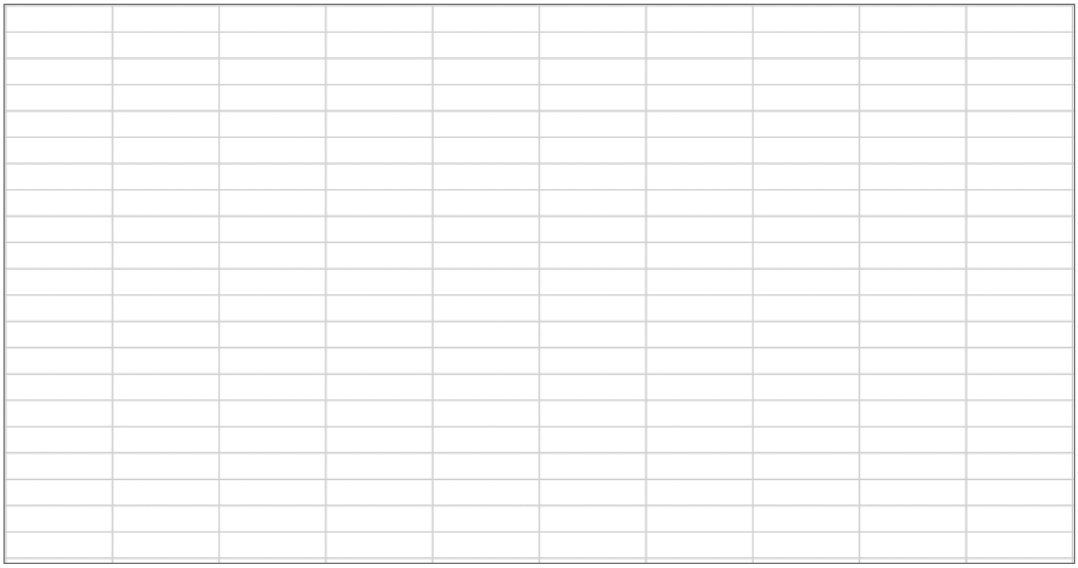

FIGURE 1.2: Multidimensional array

If we were going simple, we could define a one-dimensional array using the following declaration. It creates an array of 15 integers. Notice the way we indicate the datatype being used. Rather than using *`variable: datatype`* as we have done previously, we indicate that it's an array using the square brackets (`[]`). Inside the square brackets we include the datatype, followed by a semicolon, then the size of the array. If you wanted, you could also include a set of initial data. That could be done with a comma-separated list of values inside square brackets, such as [3, 6, 9, 14, 2, 15, 16, 90, 145]. You then have an initialized array of values. Again, without the `mut` keyword, we can't change any values once they have been set, though you don't have to set them when you declare the array. However, one thing you do need to do is make sure you have filled the array to the size you have declared.

```
let list: [i32; 15];
```

If you were to use the following code, you'd get a compiler error. The error would tell you that an array was defined with 15 elements but that only 8 elements were found. Rust expects a fixed-size array to be populated to the length of the array. If you are going to use only 8 values, you should declare an array with only 8 values. Rust sees a declaration of an array as essentially a datatype. `i32; 15` is the datatype the variable is defined as. Anything that doesn't exactly match that fails the type check.

```
let array: [i32; 15];
array = [3, 43, 12, 18, 90, 32, 8, 19];
```

In our case, we are working with a multidimensional array. If you wanted to declare a multidimensional array in C, you could use `int array[10][10]`. This says you have an array that is 10 values wide by 10 values deep. If you wanted a three-dimensional array, you would just tack on an additional number in square brackets. In Swift, it would look like `var array = Int[][]`, which is an unbounded multidimensional array. No size is specified in either direction. In Rust, we don't close the square brackets to create the additional dimension. A multidimensional array is created using the following code:

```
let array: [15]; 15];
```

This is an array where no datatype has been specified. If you want to specify a datatype, you need to initialize the array. Pick a value and then the datatype, as we did here. This means something like `[[95u16; 10]; 10]` if you want an unsigned 16-bit integer array with the value 95 placed in all the cells. The other option is to simply not declare the datatype and let Rust infer it when you initialize it for real. We'll get to one way to initialize the array momentarily.

To access array elements, you use the `[]` notation. If you wanted to get to position 5 in your array, you'd use `array[4]`, keeping in mind that arrays are 0-based, meaning you start accessing arrays starting with an index of 0. If you had an array you had defined as `[15]`, you'd access the 15 elements using the values 0–14. Trying to use `[15]` to get to a value in that array would generate an error because you would've gone beyond the defined bounds of the array.

Control Structures

Any programming language needs to have control structures. Programmers cannot live by variables and declarations and functions alone. We need things like conditionals where we compare something and make a decision based on that comparison. This might be an `if` statement, for instance. We also need loops. For the main function here, we are going to look at one type of loop, which is a `for` loop.

A `for` loop might use a counter that gets incremented each pass through the loop. When it comes to arrays, we can make use of the loop counter as an index into the array. You can see that in the following code:

```
        for i in 0..74 {
            for j in 0..74 {
                if rand::random() {
                    world[i][j] = 1;
                } else {
                    world[i][j] = 0;
                }
            }
        }
```

Let's deconstruct just one of these and then talk about why there are two here. The line is `for i in 0..74 {`. The `0..74` is a set of all integer values starting with 0 and ending with 74. The `..` indicates a range of values. Since we are going to use the variable *`i`* as an index to the array, we need to start at 0. We don't have to start at 0 just because it's a `for` loop. This would be similar to a C loop that looked like the following, which does the same thing but just expresses the range using less than or greater than:

```
    for (i=0; i<75; i++) {
```

Rust is closer to Python than it is to C when it comes to writing for loops. In Python, the same loop would look like the line that follows. In Rust, the range is more elegantly expressed with `0..75`, where Python uses the keyword `range`, which generates a range of values starting at 0, ending at the value passed to `range` and incrementing by 1 each pass through the `for` loop. The behavior is the same as the `for` loop written in Rust earlier.

```
    for i in range(74):
```

We are using nested loops, which means we have two separate `for` loops. Without the nested loops, we'd end up with a diagonal line through our two-dimensional array, because the same value would be used on the x-axis as the y-axis. In this case, we use *`i`* as our row counter and *`j`* as our column counter. For every iteration of *`i`*, we run through the entire row by running through each possible column using the *`j`* variable. Speaking of variables, the `for` statement automatically declares and creates our *`i`* and *`j`* for us. You'll also note that the `mut` keyword is implicit in the creation of the two variables, since the value has to change as we iterate over the range of values. The loop wouldn't work well if the loop index didn't iterate. Think about the C implementation of the same loop. If you left off the `i++`, which increments the index value, the loop would just keep going endlessly because the condition that keeps the loop going (`i<74`) would always be met since *`i`* never increases. It remains at 0 without that incrementing.

The heart of creating the world is inside the loops we have used. The code for that follows, and it uses random values to determine whether the cell is alive or dead in the initial generation. We call the function `random()` out of the `rand` crate, which we included at the top of the program. This function generates a Boolean. And this brings us to another control structure. We are using `if` as a decision point. If we get a true out of `rand::random()`, then we set the cell with a value of 1. Otherwise, we

set the cell with a value of 0. The `else` keyword indicates that if the first condition is not true, then the enclosed block of code is executed. Using `else` saves us from having to use another condition. The only thing we care about with an `else` statement is whether or not the first condition is true.

```
if rand::random() {
    world[i][j] = 1;
} else {
    world[i][j] = 0;
}
```

You may notice that the initial condition doesn't have parentheses around it. You will find this is common in Rust programming. In fact, the Rust compiler will let you know that you don't need them if you include them. As someone who has been writing programs in multiple languages over multiple decades, I find the use of parentheses clarifies the logic of my expression. Some languages require that you put the expression in parentheses. Rust is not one of those languages. Leave the parentheses out unless you absolutely have to have them to get the right value out of a complex Boolean expression.

To set the value of each cell in our multidimensional array, we use the two sets of square brackets to indicate the row and column of the cell. Again, we use the index values `i` and `j` to indicate where we are in the "world" we are creating.

Although this is the end of the main function in this version of the program, there are pieces missing to create a fully functional program. We have a pair of functions left to talk about, and neither of them get called. The fact that you have written code that never gets called will also generate warnings from the Rust compiler. Rust wants you to know that you should probably call the functions you have taken the time to write, just to make sure you put the functions into the right program. However, even if we aren't yet going to call these functions, we are going to move into talking about them so that you have a broader palette of colors to write your own programs with after just this chapter.

LOOKING AT MORE FUNCTION FUNCTIONS

We'll look at two additional functions for our Game of Life program. This will bring up two additional features of functions you will need to understand. The first of these is returning values from the function. This is a common feature of functions in programming languages. You don't just call a function to introduce a chunk of code, even if it's code you want to reuse. Ultimately, that function may create a new value that needs to be returned to the calling function. The calling function needs that returned value to make a decision. Of course, in order for the function to perform a meaningful task, it needs data. This means we must be able to pass data into the function so that it can act on it. We have to pass parameters into our functions, which is something we didn't do with our main function.

Returning Values

The next function we'll look at is the one called `census()`, which takes a count of all the living cells in the world. This doesn't have any direct relation to the necessary functionality for the Game of Life program, but it's a useful statistic. We want to know when our world becomes unpopulated, if it should ever get to that point. If our world did become fully unpopulated, it would probably be a good point to stop running through successive generations since it's not possible, given the rules of

the game, for cells to spring to life without any neighbors. The return value could be checked to see when it becomes 0 and the game could be stopped at that point, just as an example of the use of `census` from a pure game play perspective.

For our purposes, it gives us a chance to talk about return values. The following is the `census` function, which includes the line at the top that tells us we are returning a value. The important part from that perspective is at the end of the function declaration, `-> u16`. This tells us the function is going to return an unsigned 16-bit integer. This is a simple return type. In practice, you can return essentially any value you can make use of as a variable.

```
fn census(_world: [[u8; 75]; 75]) -> u16
{
    let mut count = 0;

    for i in 0..74 {
        for j in 0..74 {
            if _world[i][j] == 1
            {
                count += 1;
            }
        }
    }
    count
}
```

In languages like C, you have to indicate that you are returning a value using something like a `return` keyword. Rust doesn't use that. To return a value from a function, you just provide the value or variable you are returning on a line by itself at the end of the function. This is because Rust is considered an expression-oriented language. In an expression-oriented language, every construction or block is considered an expression, and as an expression, it yields a value. Because a function is an expression in Rust, it yields a value. The value the expression evaluates to for a Rust function is the last line of that function.

> ***NOTE*** *We haven't talked about an important construct as yet, but since it's not essential to developing programs, we can just drop it in here as a note. When you are writing programs, you should be commenting. This is not a necessary task, because as you can see none of the Rust code provided thus far has been commented in any way inside the code. All the comments are coming in the text of the book, so code comments seem redundant to this point. Writing comments is simple, and we use a common approach. When you want to insert a line comment, you use* `//` *and then place the comment after them. From the* `//` *to the end of the line is a comment, regardless where on the line the* `//` *are placed. You can also use* `///` *if you want to use a document comment. Using document comments, where you can use Markdown for formatting, gives you the ability to generate documentation for your project by just running* `cargo doc`*. The* `cargo` *utility creates your documentation for you, placing it in* `target/doc`*.*

Think about it this way, since you may be less familiar with expression-oriented languages. Everything you do in Rust is intended to create a result of some sort. All of the "things" you are doing—setting variables, introducing control structures, calling functions—are expressions when they return a value. Most programming languages you may be familiar with use statements. A statement doesn't return a value. In Rust, we can and do use statements. One difference between an expression and a statement is the use of the semicolon. You may have noticed that there is no semicolon at the end of the line that just says `count` at the end of the function. That's because the function is an expression and the return value is whatever is in the variable named `count`. Because expression-oriented languages are, or at least can be, different from other languages, we'll keep returning to the concept so that you can understand the differences between expression-oriented languages and statement-oriented languages.

The rest of the function provided here is fairly straightforward, especially since it includes the nested loops we've already looked at to work through the entire world, or game grid. One note, if you aren't familiar with it, is the line where we increment the number of cells that are alive in the variable `count`. We use `count += 1`, which is a shorthand way of saying `count = count + 1`. Rather than type additional characters, especially repeating the name of the variable, we just use a shorthand notation, which evaluates to the same thing. In the end, we get the same result, no matter which way we write it. Either one will work just fine. This is a way of writing incrementing variables that has been used in C for decades and has been borrowed by several other C-like programming languages.

One thing we can do in Rust that isn't possible in other languages, like C, is to return multiple values. This is done through the use of tuples. A *tuple*, speaking mathematically, is a finite ordered list. For our purposes, it's a list and it's finite. The ordered part is relevant only in the sense that you need to know which order the values are in. This isn't to say that it has to be ordered in the way that ordered often means (smallest integer to largest or alphanumeric order). What we need to be able to do is just pull the values back since we aren't naming them.

To return a value as a tuple, you can just use a comma-separated list bracketed with parentheses: `(val1, val2, val3)`. When it comes to retrieving the values on the other side, where you are calling the function you can use a tuple in the same way you did at the end of the function. Here, you can see how you'd retrieve values from a function that returned a tuple:

```
    let i: i32;
    let b: bool;
    (i, b) = function1();
```

The one other aspect of this function we didn't talk about is also in the declaration line, but we can save that for the next section.

Passing Parameters

This will be the last function for this pass at the Life program. We're going to look at how we run through the entire world to determine what cells live and die based on the rules of the game. There are a couple of ways we can go about this. This implementation assumes the world is bounded rather than wrapping around on itself. This is primarily the case because I simply can't imagine how you'd take a two-dimensional grid and connect the left end with the right end while simultaneously connecting the top and bottom. This is the problem with using Cartesian coordinates, assuming the top left of the two-dimensional array is the fixed point that everything else is relative to. This is one

reason we end up with a longish implementation, because we always have to check to see whether we are at the boundary of the grid.

```
fn generation(world: [[u8; 75]; 75]) -> [[u8; 75]; 75]
{
    let mut newworld = [[0u8; 75]; 75];

    for i in 0..74 {
        for j in 0..74 {
            let mut count = 0;
            if i>0 {
                count = count + world[i-1][j];
            }
            if i>0 && j>0 {
                count = count + world[i-1][j-1];
            }
            if i>0 && j<74 {
                count = count + world[i-1][j+1];
            }
            if i<74 && j>0 {
                count = count + world[i+1][j-1]
            }
            if i<74 {
                count = count + world[i+1][j];
            }
            if i<74 && j<74 {
                count = count + world[i+1][j+1];
            }
            if j>0 {
                count = count + world[i][j-1];
            }
            if j<74 {
                count = count + world[i][j+1];
            }

            newworld[i][j] = 0;

            if (count <2) && (world[i][j] == 1) {
                newworld[i][j] = 0;
            }
            if world[i][j] == 1 && (count == 2 || count == 3) {
                newworld[i][j] = 1;
            }
            if (world[i][j] == 0) && (count == 3) {
                newworld[i][j] = 1;
            }
        }
    }
    newworld
}
```

We're going to focus, to start with, on the function declaration, since that's where we pass parameters. However, there are some serious gotchas here that will unfold over time because they are such complex issues. Simply, to pass a parameter into a function, you essentially declare the parameter

in the function declaration. You indicate the name of the variable being passed in so that it can be referred to later by name. You also need to indicate the datatype being used.

When you are calling functions, remember that calling parameters (the things we are talking about here) are placed on the stack so the called function can access them. Local variables are also on the stack, as well as other important data. I'm bringing this up here because one of the reasons for declaring the parameter is so that the compiler knows how much space to allocate on the stack for the parameter when the function is called. Additionally, of course, the compiler needs to be able to match the declared function (its signature) with the function call. If the parameters passed in the function call don't match the function's signature, the compiler will generate an error.

Note that in the declaration we are not only taking a multidimensional array in as a parameter, we're also returning a multidimensional array. There is a reason for this. Rust is a language that is built around memory safety. Some languages use the ideas of pass by reference or pass by value. Pass by value means that the value itself is passed into the function. Pass by reference means the memory location of the data is passed into the function. Pass by value is essentially read-only. With only the value, the function can't make any changes to the data, so there are no side effects. The variable that is passed into the function is untouched when the function is done and execution is passed to the calling function.

Pass by reference allows the called function to make changes to the data because direct access to the memory location where the data is stored is provided to the called function. This would allow the called function to make changes to that memory location so that when execution passes back to the calling function, the changed value is available in that variable in the calling function. You can see a simple representation of this idea expressed in C, since the C programming language allows this type of behavior, in Figure 1.3. In this example, a variable named *x* is created that has a storage location, shown in the box in the center. Initially, that box contains the value 10. We pass the address of that box (the `&` in front of *x* indicates we are passing the address, not the value) to the function `foo`. In `foo`, we make it clear that we are getting an address by putting an `*` in front of *x*. In the function body, we dereference the variable, meaning we are assigning the value 15 to that address location.

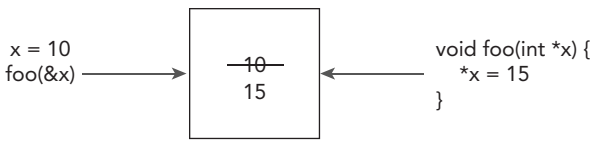

FIGURE 1.3: Passing by reference in C

This is much harder in Rust, and it will take a few passes through to explain so that you can grasp the implications. There are ways to pass values back and forth in Rust, but there is a fundamental design decision in Rust as a language that makes it a lot harder. In Rust, only one function can ever own a variable, though ownership can change hands. Before we discuss the implications of ownership in the context of this program, you will need to better understand scope.

Scope

Scope is generally an easy concept, especially since most if not all programming languages implement scope in one way or another. Scope, simply, is the space in which you can refer to a variable and have it be understood. A simplified version of one of the loops earlier would be as follows. The variable `i` here has a clearly defined scope. Anything inside the `{}` block is the scope of the variable `i`. This means we can make use of the variable `i`, and our program will happily compile and run.

```
for i in 0..74 {
    println!("{}", i);
}
```

If we try to make use of the variable outside the block of code inside the brackets, the compiler will generate an error indicating that there is no variable named `i` in the scope where it is being referred to. Here, you can see the error generated from compiling a Rust program with the `for` loop from the previous code followed by a `println!("{}", i);` statement. Not only will the code not run, it simply won't compile.

```
error[E0425]: cannot find value `i` in this scope
 --> test.rs:5:20
  |
5 |     println!("{}", i);
  |                    ^ not found in this scope

error: aborting due to previous error
```

The rules of scope aren't always straightforward, though once you learn them, they are easy enough to remember. Typically, you can say that a variable is contained in a block of code denoted by `{}`. In a function, any variable defined at the top of the function will go out of scope when the function ends. If you have a block of code that is contained within an `if` statement, as seen in the function named `generation` earlier and shown next, the brackets after the `if` statement define a scope. In the example, the code in that block is just the incrementing of the `count` variable. If the `count` variable had been defined inside those brackets, the scope of that variable would be only within those brackets. The variable passes out of scope, as we say, when the brackets close.

```
if i>0 {
    count = count + world[i-1][j];
}
```

In Rust, we have an additional complication. The complication is because of variable ownership. When we call a function with a variable as a parameter, that variable—more specifically, the memory location where the data the variable refers to is stored—becomes the property of the called function. Keep in mind that as soon as a function ends, all variables in that function go out of scope, meaning they are no longer available. In the function definition for `generation`, you can see we are passing in a variable called `world`. This is a multidimensional array. As soon as the function `generation` ends, the memory space allocated for that variable is freed automatically. Because the memory is freed, there is no way to get to the contents of that memory location any longer.

In this case, we can solve it by simply creating a whole new variable. This is done at the top of the function in the line you can see below. We create a new multidimensional array that gets populated

with the next generation of our game grid. The current generation will simply disappear once we are done calculating who is going to live, who is going to die, and who is going to be born. In our case, this is a better solution anyway, since we can't make changes to the current world without impacting the rest of the world. If we make a change to any cell in our grid, it will change the determination for subsequent cells. So, instead, we create a brand-new game grid and just swap out the existing one for the new one once we've figured out what the next generation looks like.

```
    let mut newworld = [[0u8; 75]; 75];
```

To return the new game grid back to the calling function, we place the variable on a line by itself. Most languages use an explicit return. In C, just as with many other languages, if I want to pass a value back from a function to a calling function, I use the keyword `return`, as in `return x`. Rust, instead, uses an implicit return. The last value in a function becomes the returned value. It goes on a line by itself and does not include a semicolon because it is not a statement. Instead, it's an expression. Expressions don't use semicolons to terminate them as statements do.

Because this is another complex topic in Rust, we'll continue to return to it in coming chapters. There are other ways to return values in Rust. We're just not going to address them here, so we'll save them until later.

COMPILING PROGRAMS

We have a working program at this point. Well, we have working code that needs to be compiled into a program. Rust is not an interpreted language, so we need to generate an executable. In Rust, there are two different ways to handle that task. First, Rust comes with a compiler that can be used to compile any source code. The Rust compiler is a program named `rustc`. This program can be used to generate an executable from any basic Rust source code file. This means that if you don't have any external functionality, you can compile your program with the Rust compiler and you will get an executable.

You may be familiar with some compilers that don't use the name of the program you want as the name of the output executable. Traditionally, for example, C compilers will generate a file called `a.out` when you compile without specifying an output filename. This is an artifact of the executable format that was commonly used on the Unix operating system when C compilers were first developed. Today, we don't usually use the `a.out` executable format, though many C compilers will still default to the traditional output filename. The Rust compiler will generate a file named for the file you are compiling. Here, you can see how that would work using a file named `life.rs`:

```
kilroy@milobloom:~/Documents$ rustc life.rs
kilroy@milobloom:~/Documents$ ls -la life
-rwxr-xr-x  1 kilroy  staff  288052 Jan 29 20:33 life
kilroy@milobloom:~/Documents$ file life
life: Mach-O 64-bit executable x86_64
```

However, if you used `cargo` to create your source code file and the associated directory structure, your source code file won't be named `life.rs`. It will be named `main.rs` by default. We can also use `cargo` to create the executable for us. This is generally a good habit to get into anyway since not only will `cargo` do the compilation, it will also bring in all the external functionality needed.

We'll get more into using external functionality in later chapters. For now, though, we want to build an executable from the project we have with just one source file. If we just run `cargo build` in the project directory, `cargo` will take care of doing the compiling and generating the executable, as you can see here:

```
kilroy@milobloom:~/Documents/rust/life$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.07s
kilroy@milobloom:~/Documents/rust/life$ cd target/debug/
kilroy@milobloom:~/Documents/rust/life/target/debug$ ls -la
total 1720
drwxr-xr-x@ 12 kilroy  staff     384 Jan 15 19:57 .
drwxr-xr-x   5 kilroy  staff     160 Jan  8 19:08 ..
-rw-r--r--   1 kilroy  staff       0 Dec  3 20:12 .cargo-lock
drwxr-xr-x  26 kilroy  staff     832 Dec  5 19:40 .fingerprint
drwxr-xr-x   8 kilroy  staff     256 Dec  5 19:40 build
drwxr-xr-x  56 kilroy  staff    1792 Jan 15 19:57 deps
drwxr-xr-x   2 kilroy  staff      64 Dec  3 20:12 examples
drwxr-xr-x   5 kilroy  staff     160 Dec  5 19:40 incremental
-rwxr-xr-x   2 kilroy  staff  875812 Jan 15 19:57 life
-rw-r--r--   1 kilroy  staff      99 Jan  9 20:23 life.d
lrwxr-xr-x   1 kilroy  staff      31 Dec  5 19:45 life.dSYM -> deps/life-
1a787212c1e544bc.dSYM
drwxr-xr-x   2 kilroy  staff      64 Dec  3 20:12 native
```

If you are looking closely, you will see that the dev target is what is built and the directory shown is the debug directory. This is the default build target. You can easily build the release target, which is a much cleaner build directory, shown next. Missing is all the debug information, including the file with the debug symbols in it, shown in the previous directory listing but not the one that follows:

```
kilroy@milobloom:~/Documents/rust/life/target/debug$ cd ../release
kilroy@milobloom:~/Documents/rust/life/target/release$ ls -la
total 608
drwxr-xr-x@ 10 kilroy  staff     320 Jan  8 19:08 .
drwxr-xr-x   5 kilroy  staff     160 Jan  8 19:08 ..
-rw-r--r--   1 kilroy  staff       0 Jan  8 19:08 .cargo-lock
drwxr-xr-x  17 kilroy  staff     544 Jan  8 19:08 .fingerprint
drwxr-xr-x   6 kilroy  staff     192 Jan  8 19:08 build
drwxr-xr-x  34 kilroy  staff    1088 Jan  8 19:08 deps
drwxr-xr-x   2 kilroy  staff      64 Jan  8 19:08 examples
drwxr-xr-x   2 kilroy  staff      64 Jan  8 19:08 incremental
-rwxr-xr-x   2 kilroy  staff  305392 Jan  8 19:08 life
-rw-r--r--   1 kilroy  staff     101 Jan  8 19:08 life.d
```

You may notice that both listings show an incremental directory. This is because the Rust compiler is capable of doing incremental builds to speed up the build process. In an incremental compilation, the Rust compiler builds the changes only in the source files.

We now have working source code and two ways of building our source files into executables. If you like, you can run the program that results from building it. It won't do anything interesting, but it can be executed. We have a good starting point to build additional projects on top of with the different Rust languages features we've discussed.

SUMMARY

Rust is a language that is said to be C-like, but significant differences exist between C and Rust, as there are between Rust and other C-like languages like Python. When we say C-like, what we mean is that the syntax is similar, meaning the keywords you use to cause behaviors or actions in your program are essentially the same. This includes control structures like `for` and `if`. There are some minor differences, though not significant.

One of the biggest differences between C-like languages and Rust is in variables. First of all, when we are declaring variables, we use the keyword `let` rather than just using a datatype and then the variable name. More importantly, variables cannot be changed by default. To make any changes to a variable after it has been declared, you need to declare it as mutable by using the `mut` keyword. This tells the Rust compiler that the variable can be changed later, meaning the compiler won't generate an error if it sees any attempt to change the variable.

Alongside this, an important concept in Rust is that of ownership. Remember that one of the foundational concepts in the development of Rust is memory safety. As a language that was developed knowing concurrent or parallel processing would be a possibility, those who created the Rust language determined that only one function could ever own a variable at any given time. This means that any variable essentially goes out of scope once it has been passed to another function. This doesn't mean the value goes away—just the ability to refer to the value using the alias that is the variable name.

Speaking of functions, Rust of course has functions. As you'd expect, the functions can take parameters and return values, including tuples, which are ordered collections of values. Rust can be an expression-oriented language. When it comes to programming languages, there are statements and expressions. An expression is something that evaluates to a value. A statement performs an action. A statement doesn't evaluate to a value. Even setting a variable doesn't evaluate to a value. A value gets placed into the memory location that the variable name refers to, but that's not the same as actually evaluating to a value. If you were to use a statement like `x = 10;`, the value 10 gets placed into `x`, but there is no resulting value that is residual from that.

If you are familiar with the Unix command line, you may be aware that if you were to run a program, you'd get a return value when the program was done running. This leaves a value, whether or not it gets used. This isn't a perfect analog, but it's similar. If `x = 10;` left a value that could be checked—say, a 1 or 0 indicating success or failure of the assignment—it might be considered an expression.

All of this is a long way of saying that Rust uses implicit returns through expressions. There is no `return` statement to explicitly return a value at the end of the function. Instead, you leave anything that could evaluate to a value, including a bare value or a variable, on a line by itself. Also, keep in mind that expressions do not use the semicolon to terminate the line. An expression does not get terminated like a statement does.

In the next chapter, we're going to extend Life and spend some more time looking at the ramifications of ownership on function calls and trying to reuse a variable after a function is called.

EXERCISES

1. Change the size of the grid to something other than 75×75. Keep in mind that you will need to find all the places where you have declared the life grid.

2. Call the `generation` function once. Remember that the `generation` function returns a value, so you will need to create a variable to hold the new grid that results from calling `generation`.

ADDITIONAL RESOURCES

Conway's Game of Life (from Scientific American) - `www.ibiblio.org/lifepatterns/october1970.html`

Conway's Game of Life - `pi.math.cornell.edu/~lipa/mec/lesson6.html`

What is the Game of Life? - `www.math.com/students/wonders/life/life.html`

Computer Programming - Variables - `www.tutorialspoint.com/computer_programming/computer_programming_variables.htm`

Computer Programming - Functions - `www.tutorialspoint.com/computer_programming/computer_programming_functions.htm`