The Book of V

Release 0.4.5

Peter Badida

Nov 27, 2019

CONTENTS:

The goal of this book is to teach the V language from the beginning. This book is a work in progress and I'll be editing and including new chapters as I too discover more about this new language. The book versions are released as separate Git tags containing assets hopefully satifying everyone's taste in reading formats.

See something incorrectly described, buggy or want to contribute a chapter of your own? Feel free to send a pull request and we will find a way to include it!

Feel free to buy me a coffee

CHAPTER

ONE

INSTALLATION

1.1 Prerequisities

- Internet connection
- Git (2.17.1)
- Docker (19.03.2)

Note: These are versions are mine, but older ones should work too for simple cloning and building Docker image.

1.2 Cloning, compiling and building

To ensure the same environment everywhere we'll use Docker and the default `Dockerfile` from the V repository. Current version is fetched this commit.

```
#same container that golang use
FROM buildpack-deps:buster-curl

LABEL maintainer="ANAGO Ronnel <anagoandy@gmail.com>"
WORKDIR /opt/vlang
RUN apt-get update && \
    DEBIAN_FRONTEND=noninteractive apt-get install -y --no-install-recommends gcc␣
↪clang make git && \
    apt-get clean && rm -rf /var/cache/apt/archives/* && \
    rm -rf /var/lib/apt/lists/*
COPY . .
RUN make && \
    ln -s /opt/vlang/v /usr/local/bin/v

CMD [ "bash" ]
```

Download it with cloning the repo first (currently `b51b885`):

```
git clone https://github.com/vlang/v
# or
git clone git@github.com:vlang/v
```

Then proceed to building the Docker image locally. During the building process one of the instructions is `make` which compiles V in that Docker image. Once it's built, create a container and enter it:

```
docker build --tag vlang .
docker run --interactive --tty vlang
```

Your console should now look similar to this:

```
root@16b5a9d05074:/opt/vlang#
```

The environment you entered is an isolated part of your system which contains a V installation:

```
root@16b5a9d05074:/opt/vlang# v --version
V 0.1.21 b51b885
```

By default the whole environment is isolated, but that prevents us from adding or editing files from an editor that's not installed in the Docker image. For that we will use mounting of a host directory into the container so we can:

1. Edit the files on the OS with an editor of own choice
2. Compile and run them in a consistent and reproducible environment

We will also use shortened flags for docker instead of writing e.g. `--interactive` in full.

```
docker run -i -t -v $(pwd):/opt/src -w /opt/src vlang
```

This command will run the Docker container from `vlang` image in an `interactive` mode, will allocate a `tty` for it, make visible the directory you are in (`pwd`) to the container at location `/opt/src` and change the default working directory to the project location: `/opt/src`.

Note: Once you have built the Docker image, you can navigate to any folder on your computer and run the command above. This is helpful if you have multiple projects because it'll bring the consistent environemnt with you wherever you go.

In case you make some changes to the `Dockerfile`, it's nice to have it always available even between multiple machines. You can do that with pushing and pulling Docker images.

CHAPTER

TWO

MODES

V provides multiple modes of running while the default (`v` or `v runrepl`) throws you into the REPL (Read-Execute-Print-Loop) console where you can try the language or write small scripts.

2.1 REPL

Although not in the way as it is for Python or JavaScript, this interpreter-like mode takes your code and in the background writes it to a temporary `.v` file which is then compiled and run. V then returns the output back to you in the console.

```
v  # or v runrepl
```

```
root@16b5a9d05074:/opt/vlang# v
V 0.1.21 b51b885
Use Ctrl-C or `exit` to exit
>>>
```

The console provides a simple `help` command that lists all available console commands:

```
>>> help

V 0.1.21 b51b885
  help                  Displays this information.
  Ctrl-C, Ctrl-D, exit  Exits the REPL.
  clear                 Clears the screen.
```

Note: Currently there is a hidden step between compiling `.v` file and running the final program. V requires a C compiler present on the system and attempts to compile `.v` file to `.c` which is then compiled to machine code a CPU understands. After that V runs the binary produced by the C compiler and retrieves output back.

Warning: Currently there is no support for command history, therefore arrows, Control-P, Alt-P or any combination of them will result in an escape code printed to the console.

```
>>> ^[[A
>>> ^P
>>> ^[p
>>> ^[^P
>>> ^[[1;5A
>>> ^[[1;3A
```

By default you have an access to the builtin module

```
>>> println("Hello, world!")
Hello, world!

// or even

>>> print("Hello, world!")
Hello, world!
```

In its simplest form it can be used as a calculator:

```
>>> print(1 + 1)
2
```

For more examples check Calculator test cases.

2.2 Compiler

V executable when provided with an argument that contains a `.v` suffix will open that file, compile it and produce a same-named binary.

```
// helloworld.v
println("Hello, world!")  // with a new like character \n
print("Hello, world!")    // without a new like character
```

Compile with:

```
v helloworld.v
./helloworld
```

Note: In case you can't execute the output file try changing the file into an executable with `chmod +x <file>`.

2.3 Runner

Similar to REPL mode, this mode in the background compiles and attempts to run your file.

```
v run helloworld.v
```

CHAPTER

THREE

CHAPTER I: CALCULATOR

For the first program (except Hello world of course!) let's create a simple CLI calculator. We start with a clean file in your favorite text editor:

```
// calc.v
```

3.1 Operations

We'll start with the addition. In V a function is created by `fn` keyword and a basic numeric type "integer" is noted as `int`:

```
// calc.v
fn add(left int, right int) int {
    return left + right
}
```

If we run the file:

```
v run calc.v
```

nothing happens because we are missing the entrypoint for the executable, or simply said, a place where computer should begin the execution of a program. In V the entrypoint is known as a `main` function declared as `fn main()`. Let's add a `main()` function that adds two integers and displays the output in the console. For that we'll use a built-in `println(s string)` function:

```
// calc.v
fn add(left int, right int) int {
    return left + right
}

fn main() {
    println(add(1, 2))
}
```

Now we can see the output. Let's include the rest of `+-*/` operations in the same way we created `add()` function:

```
// calc.v
fn add(left int, right int) int {
    return left + right
}

fn sub(left int, right int) int {
```

(continues on next page)

(continued from previous page)

```
    return left - right
}

fn mul(left int, right int) int {
    return left * right
}

fn div(left int, right int) int {
    return left / right
}

fn main() {
    println(add(1, 2))
    println(sub(1, 2))
    println(mul(1, 2))
    println(div(1, 2))
}
```

As we call each of the functions to perform a basic operation, we will see this output. Noticed something strange?

```
3
-1
2
0
```

`1 / 2` is apparently `0`. But why? Shouldn't the output be `0.5`?

Yes and no. For now we use `int` type which does not allow a decimal mark and values after that symbol. There is a different type that allows it written as `f32` (short for 32-bit floating-point number).

Switch all types from `int` to `f32` and run the program again.

```
// calc.v
fn add(left f32, right f32) f32 {
    return left + right
}

fn sub(left f32, right f32) f32 {
    return left - right
}

fn mul(left f32, right f32) f32 {
    return left * right
}

fn div(left f32, right f32) f32 {
    return left / right
}

fn main() {
    println(add(1, 2))
    println(sub(1, 2))
    println(mul(1, 2))
    println(div(1, 2))
}
```

```
3.000000
-1.000000
2.000000
0.500000
```

Nice, we can see a proper result for `1 / 2` operation.

3.2 Input

We can see the program computing results, but it's only for the hard-coded values directly in the `calc.v` file. This way we'd always need to rewrite our calculator.

As you've already noticed when running a V program, V already knows what file you want to use by you providing a filename in the console. The same way V can use your value for compiling we can use it for computing results. The input values can be fetched with the help of `os` module.

In V you can use a module by *importing* it via `import` keyword. From that module we will need a constant `os.args` that returns an array of another kind of V type - `string`.

```
import os

fn add(left f32, right f32) f32 {
    return left + right
}

fn sub(left f32, right f32) f32 {
    return left - right
}

fn mul(left f32, right f32) f32 {
    return left * right
}

fn div(left f32, right f32) f32 {
    return left / right
}

fn main() {
    println(add(1, 2))
    println(sub(1, 2))
    println(mul(1, 2))
    println(div(1, 2))
    println(os.args)
}
```

Now you should see even the array of arguments passed to the calculator program the same as below:

```
3.000000
-1.000000
2.000000
0.500000
["./calculator-basic-ops-float"]
```

The first argument will always be a name of the executable that's running, which in this case means `./calc`, and anything other added after the path to the executable is added as the next argument.

```
# call
v run calc.v argument
./calc argument

# output
...
["./calc", "argument"]
```

3.3 Pseudo-stack with `array`

Once we can access the console arguments, we can quickly implement so called Reverse Polish notation with a `for` loop, `if` conditional and stack.

First we skip the executable path:

```
import os

fn main() {
    for idx, value in os.args {
        if idx == 0 {
            continue
        }
        println(value)
    }
}
```

To implement Reverse Polish notation we will use an `array` as a pseudo-stack structure for storing the `f32`. To create a variable V uses a simple `<name> := <value>` syntax e.g. `number := 1`, however for an array there is a catch. We need to go a little bit further and specify the type of all values in it as `<name> := []<type>`.

```
fn main() {
    stack := []f32
    stack << 1.2  // error
}
```

After we use a keyword `mut`, we mark the variable as editable and can use << operator for the array to append a new value to it.

Currently there is no quick way for popping the last element from an array while removing it at the same time, therefore we will access the last element by its position in angle brackets (`array[idx]`) in the array and then use `array.delete()` to remove it.

```
fn main() {
    mut stack := []f32

    // push
    stack << 1.2
    stack << 2.2
    stack << 3.2

    // print the array and its length
    println(stack)
    println(stack.len)

    // pop last item
```

(continues on next page)

(continued from previous page)

```
    temp := stack[stack.len - 1]
    stack.delete(stack.len - 1)

    // print the array and the popped item
    println(stack)
    println(temp)
}
```

As you can see, an `array` has an `array.len` attribute. It's changed on each resizing manipulation of `array`.

Back to the calculator code, we will use this pseudo-stack with manually pushing and removing items to implement Reverse Polish notation from console arguments.

We'll create two array "buckets" for two categories of operators according to their precedence in an ordinary calculator.

After that let's take care of an obvious error that might be raised - calling an operator function when there isn't enough values on the stack. We need to check the number of elements in the `stack` array by its `len` attribute as we did for popping the values from it in previous example and then exit the program with a warning for which we'll use `panic(s string)`.

We can concat a variable, to a different string by using `$` symbol and a name of a variable in a string like this: `println("Value:  $my_variable")`.

```
fn main() {
    // RPN stack
    stack := []f32
    prioritized := ["mul", "div"]
    normal := ["add", "sub"]

    for idx, value in os.args {
        if idx == 0 {
            continue
        }

        println(value)
        if (value in prioritized || value in normal) && stack.len < 2 {
            panic("No values to use for operator $value")
        }
    }
}
```

Now let's append the console arguments on the stack if they are not one of the operator functions' names we added to the arrays. By default any value from the console arguments is a `string` which means we are missing one step. We need to convert the value to `f32` before appending it. By looking at the `string` implementation we can find its function for conversion `string.f32(s string)` which we should use before trying to append the value on the stack.

Once the values are converted and on the stack, we need to check if there are at least two and in the next console argument is an operator function's name pop the values into `left` and `right` variables which will be used for printing out the result.

```
fn main() {
    // RPN stack
    mut stack := []f32
    prioritized := ["mul", "div"]
    normal := ["add", "sub"]
```

(continues on next page)

(continued from previous page)

```
    for idx, value in os.args {
        if idx == 0 {
            continue
        }

        if (value in prioritized || value in normal) {
            if stack.len < 2 {
                panic("No values to use for operator $value")
            } else {
                right := stack[stack.len - 1]
                stack.delete(stack.len - 1)

                left := stack[stack.len - 1]
                stack.delete(stack.len - 1)

                if value == "add" {
                    println(add(left, right))
                } else if value == "sub" {
                    println(sub(left, right))
                } else if value == "mul" {
                    println(mul(left, right))
                } else if value == "div" {
                    println(div(left, right))
                } else {
                    println("$left $value $right")
                    println(stack)
                    panic("This should not happen!")
                }
                continue
            }
        }

        if (value in prioritized || value in normal) && stack.len < 2 {
            panic("No values to use for operator $value")
        }

        if !(value in prioritized || value in normal) {
            stack << value.f32()
        }
    }
}
```

Now we can check some basic instructions for our calculator this way:

```
// v calc.v
// ./calc 1.5 2 add
3.500000
// ./calc 1.5 2 sub
-0.500000
// ./calc 1.5 2 mul
3.000000
// ./calc 1.5 2 div
0.750000
```

Although usable, it's very limited as it does not provide any option for joined operations such as `(1 + 2) * (3 / (4 + 5))`. First we need to convert this operation into Reverse Polish notation instructions so we can appropriately edit the `main()` function.

```
(1 + 2) * (3 / (4 + 5)) = 1
1 2 + 3 4 5 + / *
1 2 add 3 4 5 add div mul

// each value is added on the stack as present
// computing starts immediately after an operator is present and the two
// closest values on the stack are popped in the reversed order i.e.
// right first, left second
1
1 2
1 2 +
3
3 3
3 3 4
3 3 4 5
3 3 4 5 +
3 3 9
3 3 9 /
3 1/3
3 1/3 *
1
= 1
```

For that we need to rework the result handling a bit and put it on stack instead of printing out right away - switch `println(operator(left, right))` to `stack << operator(left, right)` and then, after the computation is done, make sure there are no console arguments remaining. Then print the whole stack back to the console.

Note: Optimal result is having only a single element present on the stack, however it can happen that there will be an additional result if we provide more values than `operators + 1`.

Here is the complete solution. Obviously it can be optimized and refactored further, but that I've kept for you to have fun.

```
import os

fn add(left f32, right f32) f32 {
    return left + right
}

fn sub(left f32, right f32) f32 {
    return left - right
}

fn mul(left f32, right f32) f32 {
    return left * right
}

fn div(left f32, right f32) f32 {
    return left / right
}

fn main() {
    // RPN stack
    mut stack := []f32
    prioritized := ["mul", "div"]
```

(continues on next page)

(continued from previous page)

```
    normal := ["add", "sub"]

    for idx, value in os.args {
        if idx == 0 {
            continue
        }

        if (value in prioritized || value in normal) {
            if stack.len < 2 {
                panic("No values to use for operator $value")
            } else {
                right := stack[stack.len - 1]
                stack.delete(stack.len - 1)

                left := stack[stack.len - 1]
                stack.delete(stack.len - 1)

                if value == "add" {
                    stack << add(left, right)
                } else if value == "sub" {
                    stack << sub(left, right)
                } else if value == "mul" {
                    stack << mul(left, right)
                } else if value == "div" {
                    stack << div(left, right)
                } else {
                    println("$left $value $right")
                    println(stack)
                    panic("This should not happen!")
                }

                if idx == os.args.len - 1 {
                    println(stack)
                    exit(0)
                } else {
                    continue
                }
            }
        }

        if (value in prioritized || value in normal) && stack.len < 2 {
            panic("No values to use for operator $value")
        }

        if !(value in prioritized || value in normal) {
            stack << value.f32()
        }
    }
}
```

CHAPTER

FOUR

CHAPTER II: HANGMAN

After the first chapter you should know how to use keywords `fn` and `mut`, types `int`, `f32`, `string`, `array` as well as converting `string` input to `f32` and some basic `array` operations with `<<`, `array.len` attribute and `array.delete()`.

4.1 Game loop

In this chapter we will create a Hangman game that will pick a random word from a predefined array of values. As for any common game, we need to create a loop in which we check the inner state of the game and request a user input or action if necessary.

```
fn game_loop() {}

fn main() {
    game_loop()
}
```

To define what should be present in the game loop we need to check the Hangman description. So according to that, the game loop should have a player always guessing either a letter or the whole hidden word until a player makes 6 mistakes. In between the guesses the game should visibly note player's mistakes and unfold all places of a guessed letter. The game ends either by providing a correct word, guessing all letters correctly or at player's 6th mistake.

Let's define the game ending conditions first as having 6 user attempts total, wrap checking for the user attempt in an infinite loop and then stopping the game loop once the conditions are not matching. An infinite loop in V is defined as a `for` loop without any specified condition. Afterwards the loop can be broken by a `break` keyword.

```
fn game_loop() {
    max_attempts := 6
    mut attempts := 0
    for {
        attempts++
        println("User attempt $attempts")
        if attempts >= max_attempts {
            break
        }
    }
}

fn main() {
    game_loop()
    println("Game over")
}
```

We can see that the amount of attempts is a value that won't change, therefore we can declare it as a constant with `const` keyword. Unlike ordinary variables a value to a `const` is assigned via = instead of := and can't be changed.

```
const (
    max_attempts = 6
)

fn game_loop() {
    mut attempts := 0
    for {
        attempts++
        println("User attempt $attempts")
        if attempts >= max_attempts {
            break
        }
    }
}

fn main() {
    game_loop()
    println("Game over")
}
```

Next we retrieve a user input so we can. If the length of the input is `1` we will treat it as guessing a character from the whole word and display all its occurences if the character is present. If the input is longer than one character, player is guessing the whole word and only if the word matches we display it. For any other case just take it as a failed attempt.

For user input import `os`, then use its `os.get_line()` function to retrieve a single line from console - or in other words an input terminated by a single `Enter` key.

```
import os

const (
    max_attempts = 6
)

fn game_loop() {
    mut attempts := 0
    guess_word := "hangman"
    mut user_input := ""

    for {
        attempts++
        println("User attempt $attempts")

        user_input = os.get_line()
        if user_input.len > 1 {
            println("Guessing a word: $user_input")
        } else if user_input.len == 1 {
            println("Guessing a character: $user_input")
        }

        if attempts >= max_attempts {
            break
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
fn main() {
    game_loop()
    println("Game over")
}
```

Once we have the input available, let's add a sample word `hangman` to a variable. Then create a mask of that word, a value constructed of – characters in the same length as the guess word. That's easily achievable with `string.repeat(count int)` function.

```
import os

const (
    max_attempts = 6
)

fn game_loop() {
    mut attempts := 0
    guess_word := "hangman"
    display_word := "-".repeat(guess_word.len)
    mut user_input := ""

    for {
        attempts++
        println("Attempt $attempts")
        println("Word: [ $display_word ]")

        user_input = os.get_line()
        if user_input.len > 1 {
            println("Guessing a word: $user_input")
        } else if user_input.len == 1 {
            println("Guessing a character: $user_input")
        }

        if attempts >= max_attempts {
            break
        }
    }
}

fn main() {
    game_loop()
    println("Game over")
}
```

If a player guesses correctly, go through the variable which stores `hangman` word, find each occurence of the character and replace the – characters with uncovered ones according to the position in the original word.

```
import os

const (
    max_attempts = 6
)

fn game_loop() {
    mut attempts := 0
    guess_word := "hangman"
```

(continues on next page)

(continued from previous page)

```
    mut display_word := "-".repeat(guess_word.len)
    mut user_input := ""

    for {
        attempts++
        println("Attempt $attempts")
        println("Word: [ $display_word ]")

        user_input = os.get_line()
        if user_input.len > 1 {
            println("Guessing a word: $user_input")
            if user_input == guess_word {
                display_word = guess_word
            }
        } else if user_input.len == 1 {
            println("Guessing a character: $user_input")
            mut buffer := ""
            for idx, value in guess_word {
                if value == display_word[idx] {
                    buffer += display_word[idx].str()
                    continue
                }

                if value != user_input[0] {
                    buffer += "-"
                    continue
                }

                buffer += guess_word[idx].str()
            }
            display_word = buffer
        }

        if display_word == guess_word {
            println("Correctly guessed!")
            break
        }

        if attempts >= max_attempts {
            println("Game over")
            break
        }
    }
}

fn main() {
    game_loop()
}
```

Notice the sections where the `str()` function is called. While a word is stored as a `string`, that's in simple terms just an `array` of `byte` types. A `byte` on the other hand is so similar to an `int` that the `str()` function is the same for both - `int.str(c byte)`.

4.2 Improvements

If we look properly at this large game loop, we can see multiple parts that can be pulled out into separate functions which will increase the readability of the overall code. Let's move the code working with user input and name it as `guess()` function. This function will take multiple `string` parameters and also return one `string`. We reflect those properties to the function declaration and the result should look like this: `fn guess(input string, word string, mask string) string`.

```
fn guess(input string, word string, mask string) string {
    mut new_mask := mask

    if input.len > 1 {
        if input == word {
            new_mask = word
        }
    } else if input.len == 1 {
        mut buffer := ""
        for idx, value in word {
            if value == mask[idx] {
                buffer += mask[idx].str()
                continue
            }

            if value != input[0] {
                buffer += "-"
                continue
            }

            buffer += word[idx].str()
        }
        new_mask = buffer
    }
    return new_mask
}
```

For the conditions of word matching and attempts we create `check_continue()` function and make it return a `bool` type so the `for` loop can automatically check the value and exit game loop when necessary. Similarly for the match of the guessed word and already uncovered letters we create `check_win()` function so the logic is kept at one place.

```
fn check_win(word string, mask string) bool {
    return word == mask
}


fn check_continue(word string, mask string, attempts int) bool {
    return !check_win(word, mask) && attempts < max_attempts
}
```

Finally, we move out the game summary prints to `print_summary()` function, clean unnecessary variables and add spacing between code lines.

```
import os

const (
    max_attempts = 6
```

(continues on next page)

(continued from previous page)

```
)

fn guess(input string, word string, mask string) string {
    mut new_mask := mask

    if input.len > 1 && input == word {
        new_mask = word
    } else if input.len == 1 {
        mut buffer := ""

        for idx, value in word {
            if value == mask[idx] {
                buffer += mask[idx].str()
                continue
            }

            if value != input[0] {
                buffer += "-"
                continue
            }

            buffer += word[idx].str()
        }
        new_mask = buffer
    }
    return new_mask
}


fn check_win(word string, mask string) bool {
    return word == mask
}


fn check_continue(word string, mask string, attempts int) bool {
    return !check_win(word, mask) && attempts < max_attempts
}


fn print_summary(word string, mask string) {
    if check_win(word, mask) {
        println("Correctly guessed!")
    } else {
        println("Game over!")
    }
}


fn game_loop() {
    guess_word := "hangman"
    mut display_word := "-".repeat(guess_word.len)

    mut attempts := 0
    mut do_loop := true

    for do_loop {
        attempts++
```

(continues on next page)

(continued from previous page)

```
        println("Attempt $attempts")
        println("Word: [ $display_word ]")

        display_word = guess(os.get_line(), guess_word, display_word)
        do_loop = check_continue(guess_word, display_word, attempts)
    }

    print_summary(guess_word, display_word)
}

fn main() {
    game_loop()
}
```

While the current state of the game is workable, we still need to make it dynamic otherwise the game would be always the same and after the first try everyone knows the answer. We can solve this by using a file named `words.txt` which will contain guess words one per each line.

To read lines from a file on your operating system use `os.read_lines(path string)` function. Specify just the name of the file to open it from the current "working" directory or in other words, from the folder you run your program from.

After the `array` of `string` s is retrieved we need to pull a single word for the game to begin. Import `rand` and use `rand.next(max int)` from it. Make sure the maximum value is set to the length of the words `array` to only access its item by an index within the range of available words.

```
fn load_word(path string) string {
    lines := os.read_lines(path)
    return lines[rand.next(lines.len)]
}
```

If you try to include this function into the already existing game, it will most likely have a consistent behavior (always the same word chosen). That's because a randomness *seed* needs to be different on each run. For that import `time` and input `time.now()` to a call setting the seed - `rand.seed(s int)`.

Since `rand.seed(s int)` requires an `int` type we can convert the `time.Time` into a UNIX timestamp which is a number we can safely use for seeding in this particular case. `time.Time` stores it in `time.Time.uni` attribute.

Once the seed is set we can call `rand.next(max int)`. Now change the hard-coded guess word in `game_loop()` into `load_word("words.txt")` and create a file named `words.txt` in the same folder as is the hangman `.v` file. You can find a sample file in the *Appendix* section.

```
import os
import rand
import time


const (
    max_attempts = 6
)


fn load_word(path string) string {
    lines := os.read_lines(path)
    rand.seed(time.now().uni)
    return lines[rand.next(lines.len)]
}
```

(continues on next page)

(continued from previous page)

```
fn guess(input string, word string, mask string) string {
    mut new_mask := mask

    if input.len > 1 && input == word {
        new_mask = word
    } else if input.len == 1 {
        mut buffer := ""

        for idx, value in word {
            if value == mask[idx] {
                buffer += mask[idx].str()
                continue
            }

            if value != input[0] {
                buffer += "-"
                continue
            }

            buffer += word[idx].str()
        }
        new_mask = buffer
    }
    return new_mask
}


fn check_win(word string, mask string) bool {
    return word == mask
}


fn check_continue(word string, mask string, attempts int) bool {
    return !check_win(word, mask) && attempts < max_attempts
}


fn print_summary(word string, mask string) {
    if check_win(word, mask) {
        println("Correctly guessed!")
    } else {
        println("Game over!")
    }
}


fn game_loop() {
    guess_word := load_word("words.txt")
    mut display_word := "-".repeat(guess_word.len)

    mut attempts := 0
    mut do_loop := true

    for do_loop {
        attempts++
```

(continues on next page)

(continued from previous page)

```
        println("Attempt $attempts")
        println("Word: [ $display_word ]")

        display_word = guess(os.get_line(), guess_word, display_word)
        do_loop = check_continue(guess_word, display_word, attempts)
    }

    print_summary(guess_word, display_word)
}

fn main() {
    game_loop()
}
```

CHAPTER

FIVE

CHAPTER III: WORD COUNTER

Counting can be a very simple example which can help explaining the grouping of some variables under one roof. Let's jump straight into fetching console arguments via `os` and define three modes this program will work with:

- `-w` or `--words`
- `-l` or `--lines`
- `-c` or `--chars`

```
import os


fn parse_mode(args []string) string {
    mut mode := ""
    words := ["-w", "--words"]
    lines := ["-l", "--lines"]
    chars := ["-c", "--chars"]

    if args[1] in words {
        mode = "words"
    } else if args[1] in lines {
        mode = "lines"
    } else if args[1] in chars {
        mode = "chars"
    }
    return mode
}


fn main() {
    mode := parse_mode(os.args)
    println("Mode: $mode")
}
```

To generalize the mode we create a container for it - a `struct` - with `struct` keyword. The container will hold the mode's name, its console arguments and later some other attributes. By default everything stored in a `struct` is immutable and pretty much inaccessible which allows us to have efficient abstractions without hacky hot-fixes unless we explicitly allow them.

```
struct Mode {
    name string
    cli_args []string
}
```

Once the struct is declared with name, curly brackets and its attributes it's ready for usage. Although there are multiple

ways for populating its attributes with values, we'll use explicitly stated attribute names before values. This allows us to specify the attributes in an unordered way and it's quite important in the long run from the project maintainability perspective due to no requirement of keeping the order of the `struct` attributes (which may change by including a new feature).

```
import os

struct Mode {
    name string
    cli_args []string
}


fn parse_mode(args []string) Mode {
    mut mode := Mode{}
    words := Mode{name: "words", cli_args: ["-w", "--words"]}
    lines := Mode{name: "lines", cli_args: ["-l", "--lines"]}
    chars := Mode{name: "chars", cli_args: ["-c", "--chars"]}

    if args[1] in words.cli_args {
        mode = words
    } else if args[1] in lines.cli_args {
        mode = lines
    } else if args[1] in chars.cli_args {
        mode = chars
    }
    return mode
}


fn main() {
    mode := parse_mode(os.args)
    println("Mode: $mode.name")
}
```

Now we can propagate each `struct` that's initialized out of the function. Although V presents itself as not having a global space for symbols, `const` keyword creates a very similar space as global one, but on the module level. Since we can't rewrite the value assigned as a constant, in combination with `import` keyword the module space is quite a powerful and safe feature.

```
import os

struct Mode {
    name string
    cli_args []string
}


const (
    words = Mode{name: "words", cli_args: ["-w", "--words"]}
    lines = Mode{cli_args: ["-l", "--lines"], name: "lines"}
    chars = Mode{name: "chars", cli_args: ["-c", "--chars"]}
)


fn parse_mode(args []string) Mode {
    mut mode := Mode{}
```

(continues on next page)

(continued from previous page)

```
    if args[1] in words.cli_args {
        mode = words
    } else if args[1] in lines.cli_args {
        mode = lines
    } else if args[1] in chars.cli_args {
        mode = chars
    }
    return mode
}


fn main() {
    mode := parse_mode(os.args)
    println("Mode: $mode.name")
}
```

Each mode needs some kind of configuration so the counter knows what to use for distinguishing between words, lines or characters. This configuration we name `sep` as in `separator` and set it to `<space>` for words, `\n` for lines and `<empty string>` for characters. Note that the last one will count even `<space>` or `\n` as a character.

```
struct Mode {
    name string
    cli_args []string
    sep string
}


const (
    words = Mode{name: "words", cli_args: ["-w", "--words"], sep: " "}
    lines = Mode{cli_args: ["-l", "--lines"], name: "lines", sep: "\n"}
    chars = Mode{name: "chars", cli_args: ["-c", "--chars"], sep: ""}
)
```

To count we need to fetch the path from `os.args`, open the file and process its contents with a counting function that will use currently active counting mode. To open a file `os.read_file(path string)` is used which returns the file contents and also closes the file. Nevertheless, we still need to ensure such a file is present on the system with `os.file_exists(_path string)`.

```
fn count(mode Mode, path string) int {
    mut result := 0

    if !os.file_exists(path) {
        result = -1
        return result
    }

    content := os.read_file(path) or {return result}
    for item in content {
        if "" in mode.sep || item.str() in mode.sep {
            result++
        }
    }
    return result
}
```

There is one catch with the `os.read_file(path string)` function, it returns an `Option` type. This kind of

type has to be handled in your code with an `or` block that allows only specific set of keywords.

Once we handle the failing function and remove unnecessary printing to the console the program is ready and complete.

Here is a challenge for you as a reader: Currently it handles only a single file. Try to make it handle multiple files!

```
import os

struct Mode {
    name string
    cli_args []string
    sep []string
}


const (
    words = Mode{name: "words", cli_args: ["-w", "--words"], sep: [" ", "\n"]}
    lines = Mode{cli_args: ["-l", "--lines"], name: "lines", sep: ["\n"]}
    chars = Mode{name: "chars", cli_args: ["-c", "--chars"], sep: [""]}
)


fn parse_mode(args []string) Mode {
    mut mode := Mode{}

    if args[1] in words.cli_args {
        mode = words
    } else if args[1] in lines.cli_args {
        mode = lines
    } else if args[1] in chars.cli_args {
        mode = chars
    }
    return mode
}


fn count(mode Mode, path string) int {
    mut result := 0

    if !os.file_exists(path) {
        result = -1
        return result
    }

    content := os.read_file(path) or {return result}
    for item in content {
        if "" in mode.sep || item.str() in mode.sep {
            result++
        }
    }
    return result
}


fn main() {
    mode := parse_mode(os.args)
    file := os.args[2]
    println(count(mode, file))
}
```

CHAPTER

SIX

CHAPTER IV: ROLE-PLAYING GAME

In this chapter we'll create a simplified role-playing game and leverage multiple structures via `struct`. Let's start with place properties such as name and links for connecting multiple place instances between themselves. Each place will have one previous place and two places, one on the left side and one on the right side.

```
               ___ <left>
              /
<previous> ---
              \___ <right>
```

To create a link between two place structures we need references to them and use such references as values. A symbol `&` (ampersand) creates a reference for a computer memory where our place `struct` is stored. Reference is a `voidptr` type.

```
struct Place {
    name string
    left &Place
    right &Place
    previous &Place
}
```

Each reference value for a `struct` will start as a `nil` which is a pointer that stores a value `0` and for that there is a checker in V - `isnil(v voidptr)`.

Let's try to create and connect these places with code:

```
                ___ Pile of old leaves (left)
               /
Tree (start) ---
               \___ Shrubbery (right) ---
                                          \___ Bear behind Shrubbery (right)
```

Each of the place nodes can either have `nil` as a previous/next node or an initialized different place. Set the reference again by using `&` character before the symbol you want to reference, in this case an initialized place `struct`.

```
fn main() {
    mut tree := Place{name: "Tree"}
    mut pile := Place{name: "Pile of old leaves"}
    mut shrub := Place{name: "Shrubbery"}
    mut bear := Place{name: "Bear behind Shrubbery"}

    // connect tree node with its children
    tree.left = &pile
    tree.right = &shrub
    pile.previous = &tree
```

(continues on next page)

(continued from previous page)

```
    shrub.previous = &tree

    // forward-connect shrub node only
    // because it already has 'previous' set
    shrub.right = &bear
    bear.previous = &shrub

    println(&tree)
    println(tree)

    println(&pile)
    println(pile)

    println(&shrub)
    println(shrub)

    println(&bear)
    println(bear)
}
```

Running this piece of code won't yet work due to `struct` fields being immutable. To mark them as mutable use `mut` keyword with a colon (`:`) suffix. `mut:` will mark all fields after it as mutable unless it encounters some other keyword changing the `struct` field properties.

```
struct Place {
    name string
mut:
    left &Place
    right &Place
    previous &Place
}
```

Once that is fixed, we get an output like below (the memory addresses will be different). Notice the connections between each of the nodes such as Tree memory address (`0x7ffe01a7eac0` in this case) being equal Shrubbery previous node's address.

```
0x7ffe01a7eac0
{
    name: Tree
    left: 0x7ffe01a7ea90
    right: 0x7ffe01a7ea60
    previous: (nil)
}
0x7ffe01a7ea90
{
    name: Pile of old leaves
    left: (nil)
    right: (nil)
    previous: 0x7ffe01a7eac0
}
0x7ffe01a7ea60
{
    name: Shrubbery
    left: (nil)
    right: 0x7ffe01a7ea30
    previous: 0x7ffe01a7eac0
```

(continues on next page)

(continued from previous page)

```
}
0x7ffe01a7ea30
{
    name: Bear behind Shrubbery
    left: (nil)
    right: (nil)
    previous: 0x7ffe01a7ea60
}
```

The places are ready, let's create a traveler who will navigate through them. Our traveler will for now contain a single property - location - which is a reference for an already existing place. This always requires a value and must not be `nil`. We can achieve such behavior by initializing the `struct` with a value and assigning only valid values to that property.

```
struct Traveler {
mut:
    location &Place
}

fn main() {
    ...
    traveler := Traveler{location: &tree}
    ...
    println(traveler.location.name)
}
```

While assigning invalid values e.g. an incorrectly typed (or raw) value is not an error right now and we can happily do `traveler.location = 0`, it will make the program access a part of memory which it should not be allowed to do, therefore will result in a `Segmentation fault (core dumped).`

By using reference for a place and its references for the neighborhood creating a function that will move the traveler is a piece of a cake. We just need to watch out for `nil` with a short check that both accepts and returns references for places.

```
fn nonnil_or_stay(old_place &Place, new_place &Place) &Place {
    if isnil(new_place) {
       println("Nothing is there.")
       return old_place
    } else {
       return new_place
    }
}
```

With this block we can repeat the check for multiple directions we would like to travel - back, left or right - which helps with directly assigning them to traveler's location property since it should return either its old value or the new one for the desired direction if not `nil`.

The function moving our traveler will require a mutable instance of traveler `struct` and an immutable string for the direction. Mutability can again be achieved by `mut` keyword.

```
fn move(trav mut Traveler, direction string)
```

First pull some values out of the traveler instance directly and even from its references into variables for easier access and less repetitive code.

```
place := trav.location
back := place.previous
left := place.left
right := place.right
```

Then complete the function by choosing the right location for traveler property via the place checking function and stored place references,

```
fn move(trav mut Traveler, direction string) {
    place := trav.location
    back := place.previous
    left := place.left
    right := place.right

    println("Old location $trav.location.name")
    println("Trying to move $direction")

    if direction == "back" {
        trav.location = nonnil_or_stay(place, back)
    } else if direction == "left" {
        trav.location = nonnil_or_stay(place, left)
    } else if direction == "right" {
        trav.location = nonnil_or_stay(place, right)
    } else {
        println("$direction is not valid, use back, left or right")
    }
    println("New location $trav.location.name")
}
```

and replace the place properties' values printing with move functions.

```
fn main() {
    ...
    // forward-connect shrub node only because it already has 'previous' set
    shrub.right = &bear
    bear.previous = &shrub

    move(mut traveler, "back")
    move(mut traveler, "left")
    move(mut traveler, "back")
    move(mut traveler, "right")
    move(mut traveler, "back")
}
```

Here is a sample output of how it should look like.

```
Old location Tree
Trying to move back
Nothing is there.
New location Tree
Old location Tree
Trying to move left
New location Pile of old leaves
Old location Pile of old leaves
Trying to move back
New location Tree
Old location Tree
```

(continues on next page)

(continued from previous page)

```
Trying to move right
New location Shrubbery
Old location Shrubbery
Trying to move back
New location Tree
```

Any function you create with `fn` keyword can also contain a special syntax before its name - a receiver - which is in simple terms just a specification for the compiler to allow calling that function via dot-lookup (using `.` for fetching an attribute of some entity).

Using `(variable mutability Struct)` syntax let's make the function for moving accessible directly from the traveler just by moving a function argument slightly to the left in the function declaration. Also, don't forget to make the traveler mutable.

```
fn (trav mut Traveler) move(direction string) {
    ...
}

fn main() {
    ...
    mut traveler := Traveler{location: &tree}
    ...
    traveler.move("back")
    traveler.move("left")
    traveler.move("back")
    traveler.move("right")
    traveler.move("back")
}
```

CHAPTER

SEVEN

APPENDIX

1. Calculator test cases
2. Hello, world!
3. word-list

1. types
2. keywords

CHAPTER

EIGHT

INDEX

CHAPTER

NINE

INDICES AND TABLES

- search

INDEX

P

R

S

W