# Chapter 5 – Control structures

Until now we have seen that a V program starts executing in main() and sequentially executes the statements in that function. But often we want to execute certain statements only if a condition is met: we want to make decisions in our code. For this V provides the following *conditional* or *branching structures*:

**if** **else** construct

**match** construct

Repeating one or more statements (a task) can be done with *the iterative or looping structure*:

**for** construct

Some other keywords like **break** and **continue** can alter the behavior of the loop.

There is also a **return** keyword to exit from a code block.

V entirely omits the parentheses ( ) around conditions in if, match and for-loops, creating less visual clutter than in Java, C++ or C#.

## 5.1 – The if else construct

The if tests a condition, which is a boolean or logical statement. If this evaluates, to true the body of statements between { } after the if is executed, if it is false these statements are ignored and the statement following the } is executed.

**if** condition {

// do something

}

If the body is only one line, you can write it compactly as: **if** condition { // do something }

In a 2nd variant an else, with a body of statements surrounded by { }, is appended, which is executed when the condition is false; we have then 2 exclusive branches (only one of them is executed):

**if** condition {

// do something

} **else** {

// do something else

}

In a 3rd variant another **if** condition can be placed after the else, so we have 3 exclusive branches:

**if** condition1 {

// do something

} **else if** condition2 {

// do something else

} **else** {

// catch-all or default

}

The number of else if – branches is in principal not limited, but for readability reasons this should not be exaggerated. When using this form, place the condition which is most likely true first.

The { } are mandatory, even when there is only one statement in the body (some people do not like this, but on the other hand it is consistent and according to mainstream software engineering principles).

So you can’t do: if (a) statement? This is for consistency, and you won’t have to change your code if you need to have more than 1 statement.

By convention the { after the if and else are on the same line. The else if and else keywords must be on the same line as the closing } of the previous part of the structure.

Note that every branch is indented with 2 (4, or 8) spaces or 1 tab, and that the closing } are vertically aligned with the if; this is enforced by applying vfmt.

While ( ) around the conditions are not needed, for complex conditions they may be used to make the code clearer. The condition can also be composite, using the logical operators &&, || and ! , with the use of ( ) to enforce precedence or improve readability.

A possible application of this construct is the testing of different values of a variable and executing different statements in each case, but most often the **match** statement from § 5.3 is better suited for this.

Here is an example:

Listing 5.1 - if\_construct.v:

fn main() {

  a := 10

  b := 20

  if a < b {

    println('$a < $b') // => 10 < 20

  } else if a > b {

    println('$a > $b')

  } else {

    println('$a == $b')

  }

}

Listing 5.2 - booleans.v:

fn main() {

bool1 := true

if bool1 {

println('The value is true') // => The value is true

} else {

println('The value is false')

}

}

Note that it is not necessary to test: if bool1 == true , because bool1 is already a boolean value.

It is almost always better to test for true or positive conditions, but it is possible to test for the inverse with ! (not):

if !bool1

or if !(condition)

In the last case the ( ) around the condition are often necessary, for example: if !(var1 == var2), which can of course be rewritten as the shorter:

if var1 != var2

Listing 5.3 - ifelse.v:

fn main() {

first := 10

cond := 5

   if first <= 0 {

     println('first is less than or equal to 0')

   } else if first > 0 && first < cond {

     println('first is between 0 and 5')

   } else {

     println('first is 5 or greater')

   }

}

// Output: first is 5 or greater

If statements are also expressions: they can return a result to be assigned to a variable. This is shown in the following example:

Listing 5.2B - if\_expressions.v:

fn main() {

num := 777

s **:= if** num % 2 == 0 {

println('1st branch')

'even'

}

**else** {

println('2nd branch') // 2nd branch

'odd'

}

println(s) // => "odd"

}

From tetris.v: color := if g.state == .gameover { gx.Gray } else { Colors[color\_idx] }

There’s no ternary operator in V like s:=(num % 2) ? ‘even’ : ‘odd’ in V.

Some useful examples:

1. Checking if a string str is empty:

if str == '' { … }

or:

if str.len == 0 { … }

1. A function abs to give the absolute value of an integer: ?? 🡨 should be in ch 6

fn abs(x int) int {

if x < 0 {

return -x

}

return x

}

Here is a version for floating point numbers, shortened with an if expression to :

fn abs(x f64) f64 {

return if x < 0 { -x } else { x }

}

1. A function is\_greater to compare two integers: ?? 🡨 should be in ch 6

fn is\_greater(x, y int) bool {

if x > y {

return true

}

return false

}

### Compile time $if

fn main() {

  $if windows {

    println('I run on a Windows machine') // => I run on a Windows machine

  }

  $if linux {

    println('I run on a Linux machine')

  }

  $if mac {

    println('I run on a macOS machine')

  }

  $if debug {

    println('I am debugging')

  }

}

Compile time if starts with a $. Right now it can only be used to detect an OS or a –debug compilation option.

The $if statements have to come before any declarations.

## 5.2 – The match construct

What if you are testing for values and you need more than one else if clause? In that case, a match statement is often a shorter way to write this.

match runs the code block of the first case whose value is equal to the condition expression. The value of the last expression in the code block is returned.

Compared to the switch-statement in C and Java – languages, match in V is considerably more flexible (??). It takes the general form:

**match** var1 {

val1 {…}

val2 {…}

**else** {…}

}

where var1 is a variable which can be of any type, and val1, val2, … are possible values of var1; they don’t need to be constants or integers, but they must be of the same type, or expressions evaluating to that type.

The opening { has to be on the same line as the match.

Each match branch is exclusive; they are tried first to last; so place the most probable values first.

The (optional) **else** branch is executed when no value is found to match var1 with, it resembles the else clause in if-else statements; it is the last branch in the statement.

Listing 5.4 - match.v:

fn main() {

  os := 'windows'

  match os {

    'windows' { println('Running on Windows') } // => Running on Windows

    'darwin'  { println('Running on macOS') }

    'linux'   { println('Running on Linux') }

    else      { println('Running on a different OS: $os') }

  }

}

A match code block can also contain multiple statements:

  c := `c`

  mut x := ''

  match c {

        `a` {

            println('${c.str()} is for Apple')

            x += 'Apple'

        }

        `b`  {

            println('${c.str()} is for Banana')

            x += 'Banana'

        }

        `c`  {

            println('${c.str()} is for Cherry') // => c is for Cherry

            x += 'Cherry'

        }

        else  {

            println('NOPE')

        }

  }

  println(x) // => Cherry

More than one value can be tested in a branch, the values are presented in a comma separated list like: val1**,** val2, val3

Here is an example:

Listing 5.5 - match2.v:

fn main() {

  num1 := 99

  match num1 {

    98, 99  { println('It\'s equal to 98 or 99') } // => It's equal to 98 or 99

    100     { println('It\'s equal to 100') }

    else    { println('It\'s not equal to 98, 99 or 100') }

  }

}

The first branch that matches is executed and then the match-statement is complete: the break from C++, Java and C# happens but is implicit, which is certainly a big improvement.

There is also no automatic fall-through:

Listing 5.5B - match3.v:

fn main() {

  i := 0

  match i {

    0 {}

    1 { f() }

  }

}

fn f() {

  println('f is called')

}

// No output!

Match can also be used as an expression; the return value can be captured in a variable:

See match\_expression.v:

fn main() {

  number := 2

  s := match number {

    1    { 'one' }

    2    { 'two' }

    else {

      'many'

    }

  }

  println(s) // two

}

You can use match to test conditions, because these are boolean values:

See match4.v:

fn main() {

  num := 1

  match num % 2 == 0  {

    true { print('The input number is even.') }

    else { print('The input number is odd.') } // The input number is odd.

  }

}

Matching on enum values:

Each case can be indicated with the .variant syntax:

See match\_enum.v:

enum Animal {

    cat

    dog

    goldfish

    pig

}

fn makes\_miau(a Animal) bool {

    return match a {

        .cat { true }

        else { false }

    }

}

fn is\_land\_creature(a Animal) bool {

    return match a {

        .cat { true }

        .dog { true }

        .pig { true }

        else {

            false

        }

    }

}

// or like this:

fn is\_land\_creature\_alt(a Animal) bool {

    return match a {

        .goldfish { false }

        else {

            true

        }

    }

}

fn main() {

  cat1 := Animal.cat

  gf := Animal.goldfish

  println(makes\_miau(cat1)) // true

  println(is\_land\_creature(cat1)) // true

  println(is\_land\_creature\_alt(gf)) // false

}

Question 5.1: (see question5\_1.v) Give the output of the following code snippet:

fn main() {

  k := 6

  match k {

    4 { println('was <= 4') }

    5 { println('was <= 5') }

    6 { println('was <= 6') }

    7 { println('was <= 7') }

    8 { println('was <= 8') }

    else { println('default case') }

  }

}

Answer: was <= 6

## 5.3 – The for construct

V only has the **for** statement to repeat a body of statements (a code block) a number of times; this is possible because it is more flexible than in other languages. One pass through the body is called an *iteration*.

### 5.3.1 Counter-controlled iteration

The simplest form is *counter-controlled iteration*, like in for1.v:

The general format is: **for** init; condition; modif **{ }**

Listing 5.6 - for1.v:

for i := 0; i < 5; i++ {

println('This is the $i iteration')

}

Output: This is the 0 iteration

This is the 1 iteration

This is the 2 iteration

This is the 3 iteration

This is the 4 iteration

The body { } of the for-loop is repeated a known number of times, this is counted by a variable (here i).

The iteration variable I is by default mutable.

The loop starts with an *initialization* for i (i := 0); this is performed only once and is shorter than a declaration beforehand. This is followed by a *conditional check* on i (i < 5) , which is performed before every iteration: when it is true, the iteration is executed. Then a *modification* of i (i++) is performed after the iteration; this could for example also be a decrement, or + or – using a step. At that point the condition is checked again to see if the loop can continue. When the condition becomes false the for-loop stops: execution is continued after the closing }.

These are 3 separate statements which form the header of the loop, so they are separated by ; but there are no ( ) surrounding the header.

Again the opening { has to be on the same line as the for.

The iteration variable i ceases to exist after the } of the for; always use short names for it like i, j, z or ix.

!! Never change the counter-variable in the for-loop itself, this is considered bad practice!!

For-loops can be *nested,* like this:

for i := 0; i < 5; i++ {

for j := 0; j < 10; j++ {

println(j)

}

println(i)

}

What happens if we use this kind of for-loop for a general Unicode-string ?

**EXERCISES:**

Exercise 5.5: for\_character.v

Create a program that prints the following (up to 25 characters):

G

GG

GGG

GGGG

GGGGG

GGGGGG

GGGGGGG

...

1 - use 2 nested for loops

2 - use only one for loop and string concatenation

Exercise 5.7: TheFizz-Buzz problem: fizzbuzz.v

Write a program that prints the numbers from 1 to 100, but for multiples

of three print ”Fizz” instead of the number, and for the multiples of five

print ”Buzz”. For numbers which are multiples of both three and five

print ”FizzBuzz”.

(hint: use an if else inside a for loop)

Exercise 5.8: Print out a rectangle of width=20 and height=10 with the \* character: rectangle\_stars.v

Exercise 5.9: Why does the following code not work?

fn main() {

for i := 0; i < 10; i++ {

println(i)

}

println(i)

}

See for\_variable.v

### 5.3.2 Condition-controlled iteration

The 2nd form contains no header and is used for *condition-controlled iteration* ( like the while-loop in other languages )

with the general format: **for** condition **{**

**}**

You could also argue that it is a for without init and modif section, so that the ; ; are superfluous.

Example: Listing 5.8 - for2.v:

fn main() {

mut i := 5

for i > 0 {

i = i - 1

println('The variable i is now: $i')

}

}

Output: The variable i is now: 4

The variable i is now: 3

The variable i is now: 2

The variable i is now: 1

The variable i is now: 0

Exercise 5.10: Calculate the sum of all integers from 0 to 100 included: for\_sum.v

Exercise 5.11: As in the previous exercise, the sum of all integers from 0 to 100 is 5050. Why does the following code then print 2550? (for\_sum2.v)

fn main() {

  mut sum := 0

  for i := 0; i <= 100; i++ {

    sum += i

    i++

  }

  println(sum)

}

### 5.3.3 Infinite loops

The condition can be absent: like in for i:=0; ; i++ or for { } (which is the same as for ;; { } but the ; ; is removed by vfmt): these are in fact *infinite loops.* The latter could also be written as: for true { },

but the normal format is: **for {**

**}**

Example: for {

println(1)

}

A program containing such a for loop has to be stopped by the operating system.

If a condition check is missing in a for-header, the condition for looping is and remains always true, so in the loop-body something has to happen in order for the loop to be exited after a number of iterations.

This kind of loop is exited via a break statement (see § 5.4) or a return statement (see § 6.1). But there is a difference: break only exits from the loop, while return exits from the function in which the loop is coded. You can always stop a running program from outside using CTRL/C.

Always take care that the exit-condition will evaluate to true at a certain moment, in order to avoid an endless loop. For an example: see listing 5.10

A typical use for an infinite loop is a server-function which is waiting for incoming requests.

Another example is an infinite game\_loop() where the loop is stopped after a boolean turns false (see Vcasino/vcasino.v):

mut can\_play := true

for **can\_play** {

...

can\_play = … // expression or function that can make can\_play false

}

(2nd example: see ch 13 hangman.v)

### 5.3.4 The for in construct

This is the iterator construct in V and you will find it useful in a lot of contexts. It is a very elegant variation, used to make a loop over every item in a collection, like arrays and maps (see chapters 7 and 8). It is similar to a foreach in other languages, but we still have the index at each iteration in the loop.

The general format is: **for** val **in** coll { }

Or : **for** ix, val **in** coll { } if you want to have the index value.

Please note the difference between Go and V: when there is only one variable it is the value (not the index as in Go).

Also be careful: val here is a copy of the value at that index ix in the collection, so it can be used only for read-purposes, the real value in the collection cannot be modified through val (try this out!).

See for\_string.v:

fn main() {

  s := 'hello'

  for c in s {

    println(c)

  }

}

/\* Output:

h

e

l

l

o

\*/

Ranges: A range like 0..10 can only be used in for loops, like this see for\_range.v):

for i in 0..10 { println(i) }

Prints 0 to 9 on successive lines, the last value is exclusive.

If you don’t need the counter in you can just replace it with \_:

for \_ in 0..10 { println('Hello V') }

Other example: In source concurrent\_news\_fetcher.v: for \_ in 0..8

Remark: There is no equivalent for the do while-statement found in most other languages, probably because the use case for it was not that important.

Exercise 5.9: What will this loop print out ?

for i := 0; i < 5; i++ {

mut v := 0

println(v)

v = 5

}

Answer: It prints 0 on 5 consecutive lines.

Question 5.2: Describe the output of the following for-loop:

fn main() {

mut s := ''

for s != 'aaaaa' {

println('Value of s: $s')

s = s + 'a'

}

}

Answer:

Value of s:

Value of s: a

Value of s: aa

Value of s: aaa

Value of s: aaaa

We’ll see examples of using for in with arrays and maps in chs. 7 and 8

## 5.4 – break and continue

A break statement exits from the current code block or loop, and continues with the statement after the code block.

A continue statement can only be used inside a loop; it breaks the current loop short, and starts the loop again with the next iteration.

A return statement exits from the current function, and goes back to the line after the function was called, see ch 6 for more detail.

#### break

Using break, the code of the for-loop in for2.v could then be rewritten (clearly less elegant) as:

Listing 5.10 – for3.go:

fn main() {

mut i := 5

for {

i -= 1

println('The variable i is now: $i')

if i <= 0 {

break

}

}

}

Output:

The variable i is now: 4

The variable i is now: 3

The variable i is now: 2

The variable i is now: 1

The variable i is now: 0

In every iteration a condition (here i <= 0) has to be checked to see whether the loop should stop. If the exit-condition becomes true, the loop is exited through the **break** statement.

A break statement always breaks out of the innermost structure in which it occurs; it can be used in any kind of for-loop (counter, condition, and so on) , but also in a match. Execution is continued after the ending } of that structure.

In the following example with a nested loop (for4.v) break exits the innermost loop:

Listing 5.11 – for4.v:

  for i := 0; i < 3; i++ {

    for j := 0; j < 10; j++ {

      if j > 5 {

          break

      }

      print(j)

    }

    print(' ')

  }

// Output: 012345 012345 012345

A for loop with a break statement can always be made shorter by placing the inverse condition right after for, making it equivalent with the while statement in other languages. Compare the following 2 snippets with the same output:

mut factorial := 1

mut counter := 1

for {

counter++

if counter > 5 {

println(factorial) // => 120

break

}

factorial = factorial \* counter

}

println(counter) // => 6

with:

mut factorial := 1

mut counter := 1

for counter <= 5 {

    factorial = factorial \* counter

    counter++

}

println(factorial)   // => 120

println(counter)     // => 6

#### continue

The keyword **continue** skips the remaining part of the loop, but then continues with the next iteration of the loop after checking the condition,

see for example Listing 5.12 – for5.v:

for i := 0; i < 10; i++ {

    if i == 5 {

      continue

    }

    print('$i ')

}

Output: 0 1 2 3 4 6 7 8 9

5 is skipped

The keyword continuecan only be used within a for-loop.

Question 5.3: Describe the output of the following valid for-loops:

**1)**

fn main() {

mut i := 0

for {

if i >= 3 { break }

println('Value of i is: $i')

i++

}

println('A statement just after for loop.')

}

Answer: Value of i is: 0

Value of i is: 1

Value of i is: 2

A statement just after for loop.

**2)**

for i := 0; i < 7 ; i++ {

   if i % 2 == 0 { continue }

   println('Odd: $i')

}

Answer: Odd: 1

Odd: 3

Odd: 5

## 5.5 – defer

The defer { } block allows us to postpone the execution of a statement, a code block or a function until the end of the enclosing (calling) function.

defer executes a code block at the end of the scope it is in (formerly: when the enclosing function returns). This happens *after* every return and even when an error occurred in the midst of executing the function, not only at the end of the function. ( Why after every return? Because the return statement itself can be an expression which does something instead of only giving back 1 or more variables ).

defer resembles the finally-block in OO-languages as Java and C#; but a defer-call is function scoped, while a finally-call is block scoped.

In most used to free resources when they are no longer needed, such as closing a file (see ch 9).

See listing defer.v:

fn main() {

  a := 1

  b := 2

  c := 3

  defer { println(a) }

  defer { println(b) }

  println(c)

}

/\* Output:

3

2

1

\*/

The defer is a code block which can contain multiple statements: see *defer\_mult.v*

fn defer\_example() {

    mut a := f64(3)

    mut b := f64(4)

    // anything within this block won't run until the code after it has completed

**defer {**

**c := math.sqrt(a+b)**

**println('The hypotenuse of the triangle is $c')**

**}**

    // this should be executed before the statements above

    a = math.pow(a, 2)

    b = math.pow(b, 2)

    print('square of the length of side A is $a')

    println(', square of the length of side B is $b')

}

defer\_example()

/\* Output:

square of the length of side A is 9.000000, square of the length of side B is 16.000000

The hypotenuse of the triangle is 5.000000

\*/

When many defer’s are issued in the code, they are executed at the end of the function in the inverse order (like a stack or LIFO): the last defer is first executed, and so on.

fn main() {

  defer { println(0) }

  defer { println(1) }

  defer { println(2) }

  defer { println(3) }

  defer { println(4) }

}

Output:

4

3

2

1

0

For examples using defer within functions, see ch 6. § 6.3.3