**PART 2 – CORE CONSTRUCTS OF THE V LANGUAGE**

# Chapter 4 – Basic constructs and elementary data types

## 4.1. Filenames - Keywords - Identifiers

V source-code is stored in **.v** files. A source-file contains code lines, who’s length have no intrinsic limit.

*Filenames:* consist of lowercase letters, like scanner.v

Filenames may not contain spaces or any other special characters.

DON’t name your file with a keyword (like if.v or type.v) on Windows.

If the name consists of multiple parts, these are separated by underscores \_ , like high\_scanner.v or scanner\_test.v

*Indentation* *rule*: different levels in code are distinguished through TABS indentation (enforced by vfmt):

one TAB per level, ?? *1 tab equals the width of 2, 4 or 8 spaces (*you can set them up however you want in your editor)*.*

In writing code in an editor of IDE, use TABS, don’t insert spaces.

The standard convention is: 1 TAB = 2 SPACES

*Identifiers in V code:* Nearly all code objects in V code have a name.

Like all languages in the C-family, this is *case-sensitive*. Valid identifiers begin with a letter (a letter is every letter in Unicode UTF-8) or \_ , and followed by zero or more letters or Unicode digits, like: X56, group1, \_x23, i, өԑ12

The following are NOT valid identifiers: 1ab (starts with a digit), for (is keyword in V), a+b (operators are not allowed)

(??) The \_ itself is a special identifier, called the *blank identifier.* It can be used in declarations or variable assignments like any other identifier (and any type can be assigned to it), but its value is discarded, so it cannot be used anymore in the code that follows.

Sometimes it is possible that variables, types or functions have no name because it is not really necessary at that point in the code and even enhances flexibility: these code objects are called *anonymous.*

This is the set of 24 *keywords* used in V code:

(?? In Appendix with a short explanation + link to chapter, section)

**as**

**break**

**const**

**continue**

**defer**

**else**

**enum**

**fn**

**for**

**go**

**goto**

**if**

**import**

**in**

**interface**

**match**

**module**

**mut**

**none**

**or**

**pub**

**return**

**struct**

**type**

It is kept deliberately small to simplify the code-parsing, the first step in the compilation process.

They are also called reserved words, because they cannot be used as an identifier.

Programs consist out of *keywords, constants, variables, operators, types, functions, flags and attributes*.

The following delimiters are used: parentheses ( ) , brackets [ ] and braces { }.

The following punctuation characters . , ; : and … are used.

Code is structured in *statements*. A statement doesn’t end with a ; like it is customary in C-like languages: the V compiler automatically inserts semicolons at the end of statements. Statements can even be separated by spaces, like here (see *statements.v*):

fn main(){   a := 3.14 + 9.333   b := 9.333 + 3.14  println('a = $a b = $b') }

// a = 12.473000 b = 12.473000

Multiple statements written on one line do impact readability, so use that cautiously.

?? The following sections perhaps better in appendices:

Flags:

**#flag** to include linker information

**#include**  to include C header files

Attributes:

**[live]** to indicate functions for hot-code reloading

## 4.2. Basic structure and components of a V program

Global variables:

V has no global variables, because they can be changed in every part of your program, which can cause difficult to track errors.

Remarks:

- C code translated to V has to allow global variables, because these exist in C.

- Global variables can be allowed, but only with the --enable-globals flag for low level applications like kernels and drivers.

### 4.2.1 Constants

Constants contain values that can never change, they are *immutable.* Because V has no global state, only constants (const), struct types and functions can be defined at the *global* level, and are known in all other functions in the code. For example, we could define our Hello string as a constant greeting, as follows:

Listing 4.1 - hello\_world2.v:

const (

greeting = 'Hello, World from V!'

const2 = 108

)

println(greeting) // -> Hello, World from V!

One or more constants must be defined in a const declaration block surrounded by ( ) outside of any functions: it is also said that they have *module scope*.

If you only have a few constants, they can also be written as:

const (odd = 'Red' even = 'Black')

Constants in V are more useful than in many other languages, because they can store more complex values, like arrays, maps and structs, or their value can be calculated by a function (see ??).

Remark 1: If you write a constant outside of a ( ) block or inside a function:

const greeting = 'Hello, World from V!'

You get the compiler error: expected "(**"**, but got greeting

Most errors or warnings in V start with: check() …

Remark 2: Try to change a constant, you get the compiler error: `const\_name` is immutable

For example:

const (

pi = 3.14

world = '世界'

)

fn main() {

pi = 5.0

}

you get the compiler error: `pi` is immutable

Other examples: see Listing 4.6 - constants.v:

const (

  pi = 3.1415

  inch\_to\_cm  = 2.54

  greek\_greeting = 'Καλημέρα κόσμε'

  asian\_greeting = 'こんにちは 世界'

  input = '{ "name": "Frodo", "age": 25 }' // a JSON string

)

println(pi)

println(inch\_to\_cm )

println(greek\_greeting)

println(asian\_greeting)

println(input)

/\* Output:

3.141500

2.540000

Καλημέρα κόσμε

こんにちは 世界

{ "name": "Frodo", "age": 25 }

\*/

Constants in V are more flexible than in some other languages, they can be assigned more complex values (see § ??). For many uses of globals in other languages, consts will suffice. This can be useful because there is no global state in V.

V has enums (see § 4.5), so it is not necessary to emulate these with constants like in Go.

### 4.2.2 The main() function

The shortest executable V program (which does nothing) is the empty program:

fn main() {

}

(This source text only contains 16 bytes, but the optimized executable on Windows is some 65 Kb in size, on Linux it is ?? . Why is this? ??)

Even the main() function is no longer necessary for 1 file programs, provided you declare variables after the functions definitions, like in this example:

Listing 4.2 – program\_without\_main.v:

// module main // is implicit!

fn ret\_ints() (int, int) {

  return 42, 108

}

a, b := ret\_ints()

println(a) // 42

println(b) // 108

The top-level-code (here starting after the function) is be the entry-point.

If however you place the following line *before* fn ret\_ints(), you get the error:

c := 7 // error: redefinition of `c`

In the examples shown we’ll leave out main() for brevity whenever it is allowed. However, when your program contains several code files, you will have to indicate the starting point of execution by including a fn main() function in one of the files.

When your code has no starting point, like the following snippet that only contains a function:

fn add**(**left int**,** right int**)** int **{**

**return** left + right

**}**

The V compiler will alert you to this with the error: V error: function `main` is not declared in the main module

A V project is structured with *modules*, which is the same concept as a package in Go. We’ll discuss them in detail in Chapter 9. For an executable program consisting of several files, although module main is not written explicitly, all these files belong to an implicit module main.

### 4.2.3 Functions

The simplest function declaration has the format: **fn** function\_name()

Between the mandatory parentheses ( ) no, one, or more *parameters* (separated by a comma , ) can be given as input to the function. After the name of each parameter variable must come its type.

The starting point of any application (the first function called or the) is a function called main() (see § 4.2.2).

The program exits - immediately and successfully - when main() returns.

main has no parameters and no return type (in contrary to the C-family), for example, for fn main(a int) you get the compile error: fn main must have no arguments and no return values.

STYLE: main() is usually written as the top of the code file for readability.

The code in functions (the body) is enclosed between braces: { }

The last } is positioned after the function-code in the column beneath ­**f**n; for small functions it is allowed that everything is written on one line, like for example: fn sum(a, b int) int { return a + b }

The same rule applies wherever { } are used (for example: if, and so on)

Functions can also be of a certain type, this is the type of the variable(s) which is (are) returned by the function.

This type is written after the function name and its optional parameter-list, like:

fn function\_name (a typea, b typeb) type\_func

The returned variable var of type\_func appears somewhere in the function in the statement:

**return var**

A function can *return more than one variable*, then the return-types are indicated separated by comma’s, like:

fn function\_name (a typea, b typeb) (type1, type2)

Then return takes the form: **return var1, var2**

This could be used when the success (true/false) of the execution of a function or the error-message is returned together with the return value. This is not so much true in V, here option types (see ch. 12) are used.

So schematically a general function looks like:

fn function\_name(parameter\_list) (return\_value\_list) {

…

}

where parameter\_list is of the form (param1 type1, param2 type2, …)

and return\_value\_list is of the form (type1, type2, …)

When there is no return type, the return\_value\_list can be omitted.

The ( ) are only necessary when there is more than one retur type.

*Function names* follow snake\_case: they start with a small letter, and every new word in the name is preceded with an \_

The execution of a function is stopped when it’s closing } is reached, or when a returnstatement is encountered. All variables local to that function are then freed from memory. The execution of the program then continues with the line following the call of the function.

The program exits normally with code 0 ( Program exited with code 0 ); a program that terminates abnormally exits with another integer code like 1.

By now you will have seen an output like this:

[Done] exited with code=1 in 0.249 seconds

When V stops your compilation with an error.

(?? Other example of code that gives an exit code 1)

The exit function can explicitly be used to terminate a program, allowing the developer to specify the exit code for operating system.

This can be used to test successful execution of he program from a script, see *exit.v:*

fn main(){

    println('before exiting...')

**exit(1)**

    // println('after exit ???') // unreachable code

}

/\*

before exiting...

[Done] exited with code=1 in 1.645 seconds

\*/

### 4.2.4 Comments

Listing 4.3 - hello\_world3.v:

// International greetings

println('Καλημέρα κόσμε; or こんにちは 世界') // Greek or Asian greetings

/\* Output:

Καλημέρα κόσμε; or こんにちは 世界

\*/

This program illustrates the international character of V by printing UTF8-characters Καλημέρα κόσμε; or こんにちは 世界, and also the characters // and /\* \*/ used to indicate a *comment*.

?? Using UTF8 as variable name:  世界 = 'world' doesn’t yet work (Oct 29 2019)

Comments of course are not compiled.

The form that is mostly used is a one-line comment starting with // at the beginning or somewhere in a line, and ending at the next LF byte (end of line).

A multi-line or block-comment starts with /\* and ends with \*/, this is mainly used for commenting out code.

Multi-line comments can be nested:

/\* This is a multiline comment.

/\* It can be nested \*/

\*/

Listing 4.4 - nested\_comments.v:

/\*

fn main() {

/\*\*/

}

\*/

fn main() {

println('it works')

}

Output: it works

Of course multiple successive lines can also be commented out by placing // at the start of each line.

Every module should have a *module comment*, a block comment immediately preceding the module statement, introducing the module and providing information relevant to the module and its functionality as a whole. A module can be spread over many files, but the comment needs to be in only one of them.

Subsequent sentences and/or paragraphs can give more details. Sentences should be properly punctuated.

Example:

// Module superman implements methods for saving the world.

//

// Experience has shown that a small number of procedures can prove

// helpful when attempting to save the world.

module superman

### 4.2.5 Types

Variables (like constants) contain data, and data can be of different *data types,* or *types* for short*.*

A type defines the set of values for a variable, and also the set of operations that can take place on those values.

Types can be *elementary (primitive),* like an integer, a floating-point number, a boolean or a string,

or *structured (composite),* like a struct, an array or a map

or *interfaces*, which only describe the behavior of a type.

There is no type-hierarchy in V.

Struct and interface type names must start with a capital letter.

Variables are always automatically created with a value: they are initialized to empty (zero) values if no other value is given.

*Undefined values cannot occur.*

*The nil or null value does not exist in V, null reference errors cannot occur!*

Use the keyword **type** for defining your own type. Then you probably want to define a struct type (see chapter 10), but it is also possible to define an *alias* for an existing type, like in:

type IZ int

type MyFn fn(a, b int)

Listing: type\_keyword.v:

type IZ int

fn main() {

println(5) // 5

}

We say that IZ has int as its *underlying type*, this makes conversion possible (see § 4.4.3 example ??).

(?? – not yet Nov 4) If you have more than one type to define, you can use the *factored* keyword form, as in:

type (

IZ int

FZ f64

STR string

)

Every value must have a type after compilation, that is: the compiler must be able to infer the types of all values:

V is a *statically typed* language.

### 4.2.5 General structure of a V-program

The following program compiles but does nothing useful, it only serves to show the preferred structure for a V-program. This structure is not necessary, the compiler does not mind if main() or the variable declarations come last, but a uniform structure makes V code better readable from top to bottom.

All structures will be further explained in this and the coming chapters, but the general ideas are:

- Import modules if necessary

* After import: declare constants and types
* Then comes the main() function
* Then come the rest of the functions, the methods on the types first; or the functions in order as they are called from main() onwards; or the methods and functions alphabetically if the number of functions is high.

Listing 4.4 - Vtemplate.v:

import (

math

ui

)

const (

greeting = 'Hello!'

)

struct T1 {

field1 int

field2 string

}

fn main() {

  a := 42

  func1()

t := T1{}

  t.method1()

  println(a) // 42

}

fn (t T1) method1() {

  //...

}

fn func1() {

  //...

}

The order of execution (program startup ) of a V application is as follows:

1. all modules are imported in the order as indicated
2. then for every module (in reverse order) all constants are evaluated.
3. Then (2) is done for the main source file, and then main() starts executing.

### 4.2.6 About naming things

Clean, readable code and simplicity are a major goal for V development. vfmt imposes the code-style.

Names of things in V should be short, concise, evocative. Long names with underscores which are often seen e.g. in Java or Python code sometimes hinder readability.

Identifiers consist only of letters, numbers, and underscores. Multipart words in identifier names are separated with \_ : they should follow the *snake\_case* style for consistency and readability, in the future V will force it’s use (`fn fooBar` will not be allowed).

Names should not contain an indication of the module: the qualification with the module name is sufficient.

## 4.3. Variables

Variables contain data, but unlike their name suggests, their value is not always variable in V!

### 4.3.1 Introduction

The only form for *declaring* *and initializing* a variable in V is with the := declaration/initialisation operator:

age := 63

For example:

Listing 4.7 – variables1.v:

fn main() {

age := 63

println(age) // 63

}

This ensures that every variable starts out with an initial value, so that its type can be inferred by the compiler.

Contrary to most other languages, variables can only be defined *inside a function.* Becausethere is no global state, variables cannot be defined on global (module) level. So there is no var declaration like in Go.

V allows global variables only in translated C code. V knows that the code has been translated. Right now it is specified with a flag.

Error when a variable is not defined:

A variable can only be declared with :=

Using = does not define a variable, it only assigns a value to the variable:

fn main() {

age = 21

}

This gives the compiler error: undefined: `age`.

V gives you a warning for unused variables, like Go (see unused.v):

fn main() {

app := 'game'

a := 10

}

This gives the compiler *warnings*:

warning: unused.v:2:11: `app` declared and not used

warning: unused.v:3:9: `a` declared and not used

In a production build (v – prod unused.v), this becomes an *error*:

unused.v:2:11: `app` declared and not used

1| fn main() {

2| app := 'game'

^

Adding a println(app) statement is enough to remedy this.

Variables are by default *immutable* in V. If you try to change a value with the = operator, like here:

Listing 4.8 – variables2.v:

fn main() {

age := 63

age = 64

}

You get the compiler error: `age` is immutable.

How to declare mutable variables?

To declare a variable as mutable, you have to precede it with the mut keyword:

Listing 4.9 – variables3.v:

fn main() {

mut age := 63

age = 64

println(age) // 64

}

The value is changed through the = assignment operator.

You will get an error if you declare a variable as mutable, but its value is never changed! Try it and comment out the 2nd line : `age` is declared as mutable, but it was never changed.

Important: the difference between := and =  
:= is used for *declaring and initializing*, = is used for *assigning*.

The following example creates a new variable b by assigning it the value on existing variable a:

Listing 4.10 – variables4.v:

fn main() {

mut a := 42

a = 44

**b := a**

println(b) // 44

}

Also a name can only be used once. Adding the line a := 108 to the previous listing gives the error: redefinition of `a`

Declaring the type explicitly:

Because of type-inference, it is not necessary to indicate the variable’s type explicitly. However, there are cases where the value is not known at the start.

Then you have to indicate the type in the format: name type

This is the case for example in:

* struct declarations (see ch 10)

struct Person {

name string

age u8

}

* function declarations (see ch 6)

fn add(x, y int) int { … }

Important to note is that the type is written after the identifier of the variable, contrary to most older programming languages. Why was this convention chosen?

First because it is like that in Go and Rust, and because it removes some ambiguity which can exist in C declarations, for example in writing int\* a, b;

Only a is a pointer and b is not. In order to declare them both pointers, the asterisk must be repeated.

(for a longer discussion on this topic, see: <http://blog.golang.org/2010/07/gos-declaration-syntax.html>)

Secondly it reads well from left to right and so is easier to understand.

Initialization of a variable:

When a variable is declared and no other value is given, it contains automatically the default zero value for its type: 0 for int, 0.0 for float, false for bool, empty string ('') for string, memory address 0x000000000000 for pointer, zero-ed struct, and so on:

*all memory in V is initialized*.

Exrercise: swap 2 variables, see *swap.v*

Scope of variables:

A variable (or another code object like constant, type or function) is only known in a certain range of the program, called the *scope.*

Code objects declared outside of any function (in other words at the top level) have *global (or module) scope*: they are visible and available in all source files of the package.

Variables can only be declared inside a function, so they have *function scope*: they are only known in that function; the same goes for the parameters and return variables of the function.

In chapter 5 we will encounter control constructs like if and for; a variable defined inside such a construct is only known within that construct (*construct scope)*.

Mostly you can think of a scope as the *code block* ( delimited by { } ) in which the variable is declared: the variable has *local scope* within that block.

A variable can’t be declared multiple times in the same scope, giving the compiler error: redefinition of `varname`.

Listing 4.12 – redefinition.v:

// doesn't compile!

fn main() {

a := 42

a := 50

}

// test.v:3

// redefinition of `a`

Unnamed freestanding code blocks can also occur within functions, like here:

Listing 4.11 – code\_blocks.v:

fn main() {

mut a := 42

a = 44

println(a) // 44

{ // code block starts here

b := 67

println(b) // 67

a = 45

} // code block ends here

Println(a) // 45

**//** b = 9 // error: undefined `b`

}

In the previous example, variable b is not known outside of the code block in which it was declared with :=

In other words: b is out of scope after the code block, so is no longer defined

A variable that is declared can of course be further used inside a block in the same function, like variable a in the enclosed block.

Shadowing of variables, that is: using the same variable name in an enclosed scope, is also not allowed:

fn main() {

a := 0

{

a := 1 // error: redefinition of a

}

println(a)

}

Multiple assignments of variables on a single line are not allowed.

var1, var2 := 1, 2 does not work, it gives the error: assignment mismatch: 2 variables but `1` returns 1 values

(?? yet, will be implemented GitHub closed issue #46)

The blank identifier \_ cannot be used to throw away values, like the value 2 in: \_ = 2 ; this gives the error: assigning `2` to `\_` is redundant

Exercise : Correct the following code so that 67 is printed twice: (see *exercise2.v*)

fn main() {

b := 7

{

b := 67

println(b)

}

println(b)

}

First you get the message: b is immutable, so you should say mut b := 7 at declaration. But then the program runs and outputs two times 67 on consecutive lines.

You **can’t** set a variable as the last value of a code block, like this:

a := {

mut this := 1

mut that := 2

... // lot of calculations

42

}

### 4.3.2 Value types and reference types

Memory in a computer is used in programs as a enormous number of boxes (that’s how we will visualize them), called *words.* All words have the same length of 32 bits (4 bytes) or 64 bits (8 bytes), according to the processor and the operating system; all words are identified by their *memory address* (represented as a hexadecimal number like 0xf840000040).

All variables of elementary (primitive) types like int, float, bool ,string, … are *value types*, they point directly to their value contained in memory:

7

(int) i

32 bit word

Figure 4.1: Value type

Also composite types like arrays (see chapter 7) and structs (see chapter 10) are value types.

When assigning with = the value of a value type i to another variable j: j = i

a copy of the original value i is made in memory.

7

(int) i

7

(int) j

Figure 4.2: Assignment of value types

The memory address of the word where variable i is stored is given by &i (see § 4.9), e.g. 0xf840000040. Variables of value type are contained in *stack* memory.

The actual value of the address will of course differ from machine to machine and even on different executions of the same program as each machine could have a different memory layout, and also the location where it is allocated could be different.

More complex data which usually needs several words are treated as reference types.

A *reference type* variable r1 contains the address of the memory location where the value of r1 is stored (or at least the 1st word of it):

(ref) r1

address1

value of r1

(ref) r2

address1

Figure 4.3: Reference types and assignment

This address which is called a *pointer* (as is clear from the drawing, see § 4.9 for more details) is also contained in a word.

The different words a reference type points to could be sequential memory addresses (then the memory layout is said to be *contiguously*) which is the most efficient storage for computation; or the words could be spread around, each pointing to the next.

When assigning r2 = r1, only the reference (the address) is copied.

If the value of r1 is modified, all references to that value (like r1 and r2) then point to the modified content.

In V, pointers (see § 4.9) are reference types).

The values that are referenced are stored in the *heap*, which is a much larger memory space than the stack.

Memory allocation:

Memory is allocated by using the C *malloc* function with as parameter the number of bytes, for example:

fn main() {

    str := malloc(4)

    println(str) // 00000000009A23F0

}

**4.3.3 Printing**

The functions **print** and **println** are used to print output to the console. They both accept only 1 argument.

print only supports a string argument, while println supports every argument that has a str() method, which returns a string-version of its argument. println prints out its parameter followed by a newline-character \n to the standard output. The same result can be obtained with print('Hello, World from V!\n')

These functions print and println can also be applied to variables, like in: println(arr); they use the default string output-format for the variable arr.

In the following code we print out a float number, an int and a boolean value.

Listing 4.12 – printing.v:

const (

  lower\_s  = 'lower s'

)

fn main() {

a := 42

pi := 3.141592

is\_sharp := true

println(a)

println(pi)

print(pi)

println(is\_sharp)

println(lower\_s)

}

/\* Output:

42

3.141592

3.141592true

lower s

\*/

To print out an empty line, use: println('') or print('\n')

The () after println are necessary because it is a function.

Exercise 4.1:Global and local scope **-** global\_local.v:

Deduce the output of the following programs andexplain your answer, then compile andexecute them.

const (

a = 'G'

)

fn main() {

n()

m()

n()

}

fn n() {

print(a)

}

fn m() {

b := 'O'

print(b)

}

Answer - // Output:

// GOG

Remark: Functions like println and basic types including strings, arrays and maps are defined in the module *builtin,* which is always by default available.

## 4.4. Elementary types and operators

In this paragraph, we discuss the boolean and numerical data types.

Values are combined together with *operators* into *expressions*, like a \* b + c, which are also values of a certain type.

Every type has its own defined set of operators, which can work with values of that type.

If an operator is used for a type for which it is not defined, a compiler error results.

A unary operator (like – in -1) works on one value (prefix), a binary operator (like \* in e \* f) works on two values or operands (infix).

V is *strongly typed: The values on both sides of a binary operator must be of the same type*. V does not implicitly convert the type of a value (??), if necessary this must be done by an explicit conversion (see § 4.4.3).

An expression is by default evaluated from left to right.

There is a built-in *precedence* amongst the operators (see § 4.4.4) telling us which operator in an expression has the highest priority, and so gets executed first.

But the use of *parentheses* **( )** around expression(s) can alter this order: an expression within ( ) is always executed first.

() are even required in complex bool expressions: (a && b) || c instead of a && b || c

The convention is to not use operators for expensive operations (like O(n), for example array.clone).

### 4.4.1 Boolean type bool

An example: is\_alive := true

The possible values of this type are the predefined constants **true** and **false.**

Two values of a certain type can be compared with each other with the *relational operators* == and !=

These expressions produce a boolean value:

*Equality operator:* **==**

This gives true if the values on both sides are of the same type and have the same (values), false otherwise.

Example: a\_var := 10 (try the examples out in the REPL)

a\_var == 5 🡨 false

a\_var == 10 🡨 true

*Not-equal operator:* **!=**

This gives true if the values on both sides are different (values), false otherwise.

Example: a\_var := 10

a\_var != 5 🡨 true

a\_var != 10 🡨 false

V is very strict about the values that can be compared: they have to be of the same type, or if they are interfaces (see Chapter 11), they must implement the same interface type.

If one of them is a constant, it must be of a type compatible with the other. If these conditions are not satisfied, one of the values has to be converted first to the other’s type.

Boolean constants and variables can also be combined with *logical* operators ( not, and, or) to produce a boolean value.

The resultant boolean value can be tested against in conditional structures (see chapter 5).

And, or and equals are binary operators; not is a unary operator. We will use T representing a true statement, and F for a false statement.

The following are the *logical operators:*

*NOT operator:* **!** !T 🡨 false

!F 🡨 true

It turns the boolean value into its opposite.

*AND operator:* **&&**

T && T 🡨 true

T && F 🡨 false

F && T 🡨 false

F && F 🡨 false

It only gives true if both operands are true.

*OR operator:* **||**

T || T 🡨 true

T || F 🡨 true

F || T 🡨 true

F || F 🡨 false

It is true if any one of the operands is true, it only gives false if both operands are false.

The && and || operators behave in a *shortcut* way: when the value of the left side is known, and it is sufficient to deduce the value of the whole expression (false with && and true with ||), then the right side is not computed anymore. For that reason: if one of the expressions involves a long-lasting calculation, put this expression at the right side.

Like in all expressions, ( ) can be used to combine values and influence the result.

Nothing can be cast to bool.

Boolean values are most often used (as values or combined with their operators) for testing the conditions of if- , for- and match (??) -statements (see chapter 5).

STYLE: A useful naming convention for important boolean values and functions is to let the name begin with is\_ or, like is\_sorted, is\_found, is\_finished, is\_visible, so code in if-statements reads as a normal sentence. Other convention ending a boolean with ?

### 4.4.2 Numerical types

#### 4.4.2.1 ints and floats

There are types for integers and floating point numbers*.*

The bit representation is two’s complement (for more info see <http://en.wikipedia.org/wiki/Two's_complement>).

V’s numerictypes have a *fixed size* (in bits) indicated by their names:

For integers: i8 (-128 -> 127)

i16 (-32768 -> 32767)

int ([− 2,147,483,648  ->  2,147,483,647](http://en.wikipedia.org/wiki/2147483647_(number))); i32 does NOT exist

i64 (− 9,223,372,036,854,775,808 -> 9,223,372,036,854,775,807)

i128 (??)

For unsigned integers: byte (0 -> 255); u8 does NOT exist

u16 (0 -> 65,535)

u32 (0 -> 4,294,967,295)

u64 (0 -> 18,446,744,073,709,551,615)

u128 (??)

For floats: f32 (+- 1O-45 -> +- 3.4 \* 1038 )

(IEEE-754) f64 (+- 5 \* 10-324 -> 1.7 \* 10308 )

A float type on its own does not exist

Examples:

n := 7 // an int

a := u64(0) which is in fact a conversion to type u64.

mut age := f32(20) see float.v)

int is the default integer type. It is always a 32 bit integer, unlike in Go and C.

f64 is the default for floating point numbers.

A f32 is reliably accurate to about 7 decimal places, a f64 to about 15 decimal places. Due to the fact that perfect accuracy is not possible for floats, comparing them with == or != must be done very carefully: f32/f64 comparison now uses machine epsilon by default.

The initial (default) value for integers is 0, and for floats this is 0.0

Floating-point constants are of type f64. Use f64 whenever possible, because all the functions of the math package expect that type.

Numbers may be denoted in octal notation with a prefix of 0 (like 077), hexadecimal with a prefix of 0x (like 0xFF) or scientific notation with e, which represents the power of 10 (e.g.: 1e3 = 1000 or 6.022e23 = 6.022 x 1023).

The module math.complex from the standard library implements working with *complex numbers*.

### 4.4.3 Mixing of types and conversions

Because V is strongly typed, *mixing of types is not allowed*, as in the following program. But constants are considered to have no type in this respect, so with constants mixing is allowed.

Listing 4.8 - type\_mixing.v (does not compile!):

fn main() {

a := u32(15)

mut b := i8(0)

b = a + a // compiler error: cannot use type `u32` as type `i8` in assignment

   n := i16(34)

   mut m := i32(0)

   m = n // error: cannot use type `i16` as type `i32` in assignment

}

The compiler error in the first example is: cannot use type `u32` as type `i8` in assignment

Likewise even an i16 cannot be assigned to an i32 (smaller to wider type), there is no implicit casting.

To avoid this, explicit conversion are done as in the following program conversions.v:

Listing 4.9 - conversions.v:

fn main() {

a := u32(15)

   mut b := i8(0)

   b = i8(a + a)

   println(b) // ok on Linux, Windows ??

n := i16(34)

mut m := i32(0)

// m = n // error: cannot use type `i16` as type `i32` in assignment

m = i32(n)

println('16 bit int is: $n')

println('32 bit int is: $m')

   c := 5.23

   d := int(c) // convert the float number a to an integer number

   println(d)  // 5

}

/\* Output:

30

16 bit int is: 34

32 bit int is: 34

5

\*/

Boolean values cannot be converted to int:

println(int(true))  gives cannot cast `bool` to `int`

As we saw in the last example, a conversion like a32bitInt **= int(**a32Float**)** truncation of the decimal part occurs, for example 3.14159 becomes 3. In general, information is lost when converting from a wider to a smaller type, therefore in order to avoid loss of accuracy always convert to a bigger numerical type.

Conversion of types in general:

If necessary and possible a value can be *converted (cast, coerced)* into a value of another type. V never does *implicit* (automatic) conversion, it must be done *explicit* as follows, with the syntax like a function call (a type like typeB is here used as a kind of function):

**value\_of\_typeB = typeB(value\_of\_typeA)**

Question 4.1: Are int and i64 the same type on a 64 bit computer ? No, int is always a signed 32 bit integer

However: string(number) and int(‘string’) do not work !

Overflowing of numbers: Apr 4 ‘19: *Overflows* are not handled right now.

No error is generated when an *overflow* occurs during an operation: high bits are simply discarded.

Constants can be of help here, they have an overflow check during compilation:

byte(1000) gives error: constant `1000` overflows `byte`

### 4.4.4 Bit operators

They work only on *integer* variables having bit-patterns of equal length:

*Binary*: Bitwise and: **&**

bits in the same position are and-ed together, see AND-operator in §4.4.1, replacing T (true) by 1 and F (false) by 0:

1 **&** 1 🡨 1

1 **&** 0 🡨 0

0 **&** 1 🡨 0

0 **&** 0 🡨 0

Bitwise or: **|**

bits in the same position are or-ed together, see OR-operator in §4.4.1,:

1 **|** 1 🡨 1

1 **|** 0 🡨 1

0 **|** 1 🡨 1

0 **|** 0 🡨 0

Bitwise xor: ^

bits in the same position are taken together according to the rule: 1 ^ 1 🡨 0

1 ^ 0 🡨 1

0 ^ 1 🡨 1

0 ^ 0 🡨 0

*BitShift:*

Left Shift: **<<** for example: bitP << n , types: integer << unsigned integer

the bits of bitP shift n positions to the left, the empty positions on the right are filled with 0’s

if n is 2, the number is multiplied by 2, left shift by n effects to a multiplication by 2n

So 1 << 10 // equals 1 KB (1024) (kilobyte)

1 << 20 // equals 1 MB (megabyte)

1 << 30 // equals 1 GB (gigabyte)

Right Shift: **>>**  for example: bitP >> n, types: unsigned integer >> integer

the bits of bitP shift n positions to the right, the empty positions on the left are filled with 0’s

if n is 2, the number is divided by 2, right shift by n effects to a division by 2n

Example (from base64):

str[j] = n >> 16

str[j] = n >> 8 & 0xff

When the result is assigned to the first operand, they can also be abbreviated like

a <<= 2 (the same as a = a << 2 )

or b ^= a & 0xffffffff (the same as b = b ^ a; b = b & 0xffffffff)

Example: bitwise.v

const (

  foo = 1

  bar = 2

)

println(foo | bar)  // 3

println(foo &bar)   // 0

println(foo ^ 2)    // 3

### 4.4.5 Logical operators

Here we have the usual ==, != (see § 4.5.1) and <, <=, > and >= working on number types, but also on strings.

They are called logical because the result value is of type bool: b3 := 10 > 5 // b3 is true

// variations:

  sum := 3

println(sum == 3) // true

  println(int(sum == 3)) // 1

### 4.4.6 Arithmetic operators

The common binary operators +, - , \* and / exist for both integers and floats.

In contrast to the general rule, this could be called a form of operator overloading; moreover the + operator also exists for strings.

/ for integers is integer division, for example: 9 / 4 gives 2.

The modulus operator % is only defined for integers: 9 % 4 gives 1

Integer division by 0 causes the program to crash, a *run-time panic* occurs (in many cases the compiler can detect this condition.

println(9 / 0)

Gives:

division or modulo by by zero

Division by 0.0 with floating point numbers gives an infinite result: **inf**

Exercise 4.4: Try this out: divby0.v

There are shortcuts for these operations: a = a + b can be shortened to a += b , and the same goes for -=, \*=, /= and %= .

In particular:

i += 1 is short for i = i + 1

i -= 1 is short for i = i – 1

(i++ and i—don’t exist)

### 4.4.7 Random numbers

This functionality comes from the rand module, which has to be imported.

From tetris.v: **rand.seed(s)**

g.tetro\_idx = **rand.next**(B\_TETROS.len)

See example: generate\_random.v:

The seed s is an arbitrary number, mostly taken from the current time.

import (

  rand

  time

)

fn main() {

  t := time.now()

  s := t.calc\_unix()

  rand.seed(s)

// generate a random number from 1 to 100

  j := rand.next(100)

  println(j)  // e.g. 23

}

The seeding can be written shorter as: rand.seed(time.now().uni)

### 4.4.8 Operators and precedence

Some operators have higher priority (precedence) than others; binary operators of the same precedence associate from left to right. The following table lists all operators and their precedence (much shorter and clearer than in C or Java), top to bottom (5 -> 1) is highest to lowest:

**Precedence Operator(s)**

**5 \* / % << >> &**

**4 + - | ^**

**3 == != < <= > >=**

**2 &&**

**1 ||**

It is of course allowed to clarify expressions by using ( ) to indicate priority in operations: expressions contained in ( ) are always computed first.

For some struct types operator overloading can be defined to enhance the readability of code (see § 9.6.3).

### 4.4.9 Aliasing types (named types) ??

When working with types, a type can also be given another name, so that this new name can be used in the code. This can have the following advantages:

* shortening a long type-name (which would otherwise lead to cluttered code)
* avoiding a name-clash (when variables from different modules have the same name and you don’t want to use the module prefix).
* to clarify the meaning of the code, thus enhancing the readability.

In type TZ int, TZ is declared as a new name for the int type (perhaps it represents time zones in a program), and can then be used to declare int variables, like in the following program:

Listing 4.11 - type.v: // does not yet compile Apr 7 ’19 on Windows, devise a better example with struct or array, later?

type TZ int

fn main() {

println(5)

}

In fact this alias is a brand new type, which can have methods that the original type does not have (see Chapter 10), for example: TZ can have a method to output the time zone-info in a clearer or prettier way.

?? Exercise 4.5: Define an alias type Rope for string and declare a variable with it.

?? Exercise 4.6: temperatures.go

Write a program that has methods to convert from Celsius to Fahrenheit and vice versa; define special types for these temperature-scales.

### 4.4.10 Unicode characters

Characters are not a distinct type in V: they are a special case of integers.

*Character literals* are surrounded by backticks ` `,

for example  `=`

str[0] = `\0`

char := `a`

int(`A`) == 65

int(str[0])

See also: chars.v

fn char\_example() {

    a\_char := `a`

    println('The ascii value of this char is: $a\_char') // => The ascii value of this char is: 97

    println('The char is: ${a\_char.str()}') // => The char is: a

    mut concat := 'b' + a\_char.str() + 'dnews be' + a\_char.str() + 'rs'

    print(concat) // => badnews bears

    // use += to append to a string

    concat += '\_appended'

    println(', $concat') // => , badnews bears\_appended

}

char\_example()

$ returns the ASCII value, use method .str() to turn a char into a string.

The byte type represents 8 bit unsigned integers, and this is ok for the traditional ASCII-encoding for characters (1 byte).

There is also support for Unicode (UTF-8): *characters* are also called *Unicode* *code points*, and a Unicode character is represented by an int (4 bytes) in memory. In documentation they are commonly represented as U+hhhh, where h is a hexadecimal digit: rune is the type which represents a Unicode code point in V.

To write a Unicode-character in code, preface the hexadecimal value with \u or \U.

Because they need at least 2 bytes we have to use the i16 or int type (??).

If 4 bytes are needed for the character \U is used; \u is always followed by exactly 4 hexadecimal digits and \U by 8 .

Here is an example displaying runes: (see rune.v)

fn main() {

  // GRINNING FACE😀 => f0 09 98 80

  grinning\_face := rune(0xf09f9880)

  println(grinning\_face)

  // COMMERCIAL AT@ => 0x40

  commercial\_at := rune(0x40000000)

  println(commercial\_at)

}

?? doesn’t show on Windows cmd or Powershell, try Linux ?

Apr 7 ’19: ch := '\u0041' // You just found a bug…

(?? When working: remake char.go)

## 4.5 Enums

(take example from Rust or Crystal enums)

V has a built-in enum type, just like Swift and ??

The enum is a type, so its name starts with a capital letter. The enum values are separated by spaces or new-lines, and surrounded by { }, and their names must also be snake\_case.

Listing 4.11B - enums.v:

enum Color {

              red green blue

}

enum Days {

        sunday

        monday

        tuesday

        wednesday

        thursday

        friday

        saturday

}

enum GameState {

        paused running gameover

}

fn main() {

        mut color := Color.red

        color = .green

        // `Color` needs to have method `str() string` to be printable

        // println(color)

        // println(Color.blue)

println(color in [.blue, .green, .red]) // true -- ?? should come in ch 7

        mut day := Days.friday

        day = .saturday

}

Since V knows that color is a `Color`, there’s no need to type `Color.green` instead of just `.green`

?? You can see that internally they are given successive integer values, starting from 0.

?? They have a built-in str() method. Nov 2: not yet, because they can’t be printed; doesn’t yet work on Windows, docs example doesn’t work

We see that you can also write the enum like:

enum Color {

red

green

blue

}

This can be more readable if the number of values is larger.

## 4.6. Strings

Strings are delimited preferably by ' ' , but double quotes "" are also allowed (Double quotes are converted to single quotes by vfmt for consistency. Unless the string contains ').

Strings are a sequence of UTF-8 characters (the 1 byte ASCII-code is used when possible, a 2-4 byte UTF-8 code when necessary). UTF-8 is the most widely used encoding, the standard encoding for text files, XML files and JSON strings. While able to represent characters that need 4 bytes, ASCII-characters are still stored using only 1 byte.

(??) A V string is thus a sequence of *variable-width* characters (each 1 to 4 bytes, see Ex. 4.6), contrary to strings in other languages as C++, Java or Python that are fixed-width (Java uses always 2 bytes). The advantages are that V strings and text files occupy less memory/disk space, and since UTF-8 is the standard, V doesn’t need to encode and decode strings as other languages have to do.

V also optimizes strings by using a pool instead of allocations for small strings.

Strings are *value* types and *immutable*: once created you cannot modify the contents of the string. Formulated in another way: strings are immutable arrays of bytes. Because of this, the substring function is very efficient: no copying is performed, no extra allocations required.

country := 'China'

  // country = 'Italy' // `country` is immutable

  mut country2 := 'China'

  country2 = 'Italy'

To change a string, prefix it with mut. But in fact, a new string is made in the last line above.

String literals: several kinds exist:

*Interpreted strings:* surrounded by ' ' (single quotes) or " " (double quotes)

escape sequences are interpreted:

for example: \n represents a newline

\r represents a carriage return

\t represents a tab

\u or \U Unicode characters

the *escape character* \ can also be used to remove the special meaning of the following character, so \" simply prints a ", and \' is ' , \\ prints a \

println('\\') // => \

str3 := 'This is Ivo\'s tutorial on V lang'

println(str3) // 'This is Ivo's tutorial on V lang'

V strings allow newlines:

a := 'a

b

c'

quotes are escaped by backslashing (\) them

*Raw strings:* prefix single quotes with r: r'raw\_string'

Eqcape sequences are not interpreted:

Example: in r'This is a raw string \n' \n is not interpreted but taken literally:

>>> println(r'This is a raw string \n')

This is a raw string \n

They can span multiple lines.

?? *C string literals:* cstr := c'hello'

Strings are length-delimited and do not terminate by a special character as in C/C++

The initial (default) value of a string is the *empty string* '' .

The usual comparison operators (== != < <= >= >) work on strings by comparing byte by byte in memory.

The length of a string str (the number of bytes) is given by str.len: (see Listing 4.12 - strings.v: )

println('abc'.len) // 3

Here are the lengths of some Unicode strings:

str0 := 'greek'

  str := 'Καλημέρα κόσμε'

  str2 := '世界'

  str3 := 'こんにちは'

  println(str0.len)   // 5

  println(str.len)    // 27

  println(str2.len)   // 6

  println(str3.len)   // 15

Converting to a string: str()

Most built-in types have a str() method, which converts a variable of this type to a string:

n := 108

println(n.str()) // 108 - – same as println(n)

pi := 3.14159

println(pi.str()) // 3.141590 – same as println(pi)

How to convert a string to an integer: use the .int() function on the string as receiver

a := '1'

b := '2'

println(a.**int()** + b.int()) // 3

How to convert a string to a float: use the .f32() or .f64() function on the string as receiver

Bytes to string conversion:

Instead of `tos2(bytes)` use the casts: string(bytes), string(bytes\_buffer, len) and string(bytes\_array).

?? example

A string is a read-only array of bytes:

The contents of a string (the ‘raw’ bytes) is accessible via *standard indexing methods*, the index is written between [ ], with theindex starting from 0:

the first byte of a string str is given by: str[0]

the i-th byte by: str[i]

the last byte by: str[str.len - 1]

str4 := 'Tokio'

  println(str4[0])  // T

  i := 3

  println(str4[i])  // i

  println(str4[str4.len - 1]) // o

However these translate to real characters if only ASCII characters are used!

The array-range notation is used also to take a substring from a string str4[n..m] (see § 4.7 Finding a substring)

?? should come in chapter 7

Note: Taking the address of a character in a string, like &str[i], is illegal.

Checking that a string contains only numbers:

Each character of the string is checked with is\_digit()

fn str\_is\_nbr(s string) bool {

for i := 0; i < s.len; i++ {

if !s[i].**is\_digit()** {

return false

}

}

return true

}

String substitution (interpolation):

A string literal can contain variable names preceded by a $ sign. When these are printed out, the variable value is substituted in the string. $ does automatic casting to string

This works for any type of variable. In this way, complex strings can be built up programmatically.

This also works for expressions, but these have to be delimited by { }, so as to execute them first.

Then they can be substituted with the $ operator: ${expression}

Listing 4.13 - substitution.v

const (

  lower\_s  = 'lower s'

)

fn main() {

  person := 'Harris'

  println('Hello, $person!')

  println('lower\_s = $lower\_s')

  pi := 3.141592

  is\_sharp := true

  print('$pi')

  println('pi is $pi')

println('pi with 2 decimals is **${pi:.2f}**') // formatting the output

  println('is\_sharp is $is\_sharp')

  a := 5

  b := 7

  println('The sum is ${a + b}')

}

/\* Output:

Hello, Harris!

lower\_s = lower s

3.141592pi is 3.141592

pi with 2 decimals is 3.14

is\_sharp is 1

The sum is 12

\*/

A useful example is building a filepath with variables, for example (see faker module ch 14)

 path := './data/$generator/$method'

Exercise:

Print out the following string with substitution: '$:USD'

Print out the following constants:

const (

USD\_VALUE = 1.0

YEN\_VALUE = 108.0

DYNAMIC\_DISPLAY\_STR = '$$USD\_VALUE = ¥$YEN\_VALUE'

)

(see print\_dollar.v)

Printing every character in a string: see § 5.3.4

Adding (concatenating) strings: **+**

Two (or more) strings s1 and s2 can be made into one string s with: s := s1 + s2

s2 is appended to s1 to form a new string s.

The shorthand += can also be used for strings.

If you want to concatenate to the string itself, it must be declared as mut:

Listing 4.14 - concatenation.v

fn main() {

mut s := 'Hel' + 'lo, '

s += 'world!'

println(s) // Hello, world!

}

`+` must have the same types on both sides. In the example below println('n = ' + n) doesn’t work: it gives the compiler error: expected type `string`, but got `int`.

n := 7

// println('n = ' + n) // compile error

println('n = $n') // 1) n = 7

println('n = ' + n.str()) // 2) n = 7

Here are two alternatives to remedy this:

1) The simplest way is to use string interpolation: println('n = $n')

2) Or you can call explicitly the str() method on n to convert it to string before adding it to another string:

Exercise: Correct the following code so it compiles:

j := 3

//i := j + ' idiots' // expected type `int`, but got `string`

Answer:

i := j.str() + ' idiots' // cast int to string using str()

 println(i) // => 3 idiots

Remark:

Concatenating many strings or in a loop structure using + is not the most efficient way, a better approach is to use strings.Builder, even better is to use writing in a byte-buffer array (§ 7.2.6).

Listing 4.14B – string\_builder.v

import strings

fn main() {

    // 1 million \* 12 byte -> a line is (line word, space, middle\_number\_of\_chars)

    mut builder := strings.new\_builder(12000000)

    for i in 0..1000000 {

        builder.write('line $i\n')

    }

print(builder.str())

}

Question: What does the following program print?

fn main() {

a := 'test'

println('a' + ' $a')

}

Answer: a test

How the string type is defined in the builtin module: see § 10.3

C strings can be converted to V strings with string(cstring) or string(cstring, len).

In the next section we learn a number of useful methods for working with strings.

## 4.7. The builtin string module

Strings are a basic data structure, and every language has a number of predefined functions for manipulating strings. In V these are gathered in the modules *builtin.string*, *strings and strconv.*

Some useful functions are:

Listing 4.15 - strings\_functions.v:

trim\_space removes all spacelike characters left and right from a string:

x2 := ' Eagle  \n'

println('\*\*' + x2.**trim\_space**() + "\*\*") // \*\*Eagle\*\*

The functions trim\_left and trim\_right are also available.

to\_lower and to\_upper respectively lowercase and uppercase a string:

// Lower case of a string using to\_lower():

  x3 := 'Eagle'

  println(x3.**to\_lower**())  // => eagle

  // Upper case of a string using to\_upper():

  println(x3.**to\_upper**())  // => EAGLE

Finding a character or substring:

The index() function gives its starting index, but it returns an option type ?int (see chapter 10), so an or block must be used to handle the case when the character is not found in the string:

// Finding a character or substring

  ix := x2.index('g') or {

    println('char not found')

    return

  }

  println(ix) // => 2

  ix2 := x2.index('ag') or {

    println('substring not found')

    return

  }

  println(ix2) // => 1

Getting a substring: string[start..end] (end character is not included), because a string is also an array of bytes

city := 'Tokio'

println(city[1..3]) // => "ok"

x3 := 'Flower'

println (x3[0..4]) // => Flow

Splitting a string: (should be in ch 7)

p := 'There are many words in a sentence'

  arr := p.**split**(' ') // split by space

  println(arr) // => ["There", "are", "many", "words", "in", "a", "sentence"]

  println(arr.len) // => 7

  // you can also split on a word

  arr2 := p.split('r')

  println(arr2) // => ["The", "e a", "e many wo", "ds in a sentence"]

split\_into\_lines can be used to split a text into sentences separated by newline characters (see ch 14 faker module)

Check if some text occurs in a string using contains:

// Check if some text occurs in a string using .contains:

  // contains true if there is a match, false if no match is found

  i4 := 'Hi, how may I help you today?'

  println(i4.**contains**('help')) // => true

  // .contains is case-sensitive:

  println(i4.contains('Help')) // => false

In the same way you can use starts\_with ends\_with to check if a string starts or ends with a certain character or word.

Replacing part of a string:

  mut x := 'There are two monkeys in the tree'

  x.**replace**('two', 'three') // replace is not in place!

  println(x)  // => There are two monkeys in the tree

  x = x.replace('two', 'three')

  println(x)  // => There are three monkeys in the tree

  // if a match is not found, it doesn't do anything, no error:

  y := 'There are mony birds in the tree'

  println(y.replace('many', 'two')) // => There are mony birds in the tree

Finding text between two characters:

  a := 'This is so [Crazy]'

  println(a.**find\_between**('[',']')) // => Crazy

  // Finding text between two words:

  c := 'This is the scraped content: <title>Welcome to V lang</title>'

  println(c.find\_between('<title>','</title>')) // => Welcome to V lang

  // if there is no match, no error:

  println(c.find\_between('<random>','</random>')) // =>

  // if there is a match with the starting string

  println(c.find\_between('<title>','</random>')) // => Welcome to V lang</title>

The method **index\_after** can also be used to get the start and end position for a string that you want to cut out of another string, see ch 13 links\_scraper.v

## 4.8. Times and dates

?? should perhaps better be in a chapter on vlib

Here is code to get the system time (*system\_time.v*):

import time

fn main() {

    t := time.now()

    // String representation

    println(t)

    // Unix time (number of seconds that have elapsed since 1970-01-01)

    println(t.uni)

    // There are different formats

    // YYYY-MM-DD HH:MM:SS

    println(t.format\_ss())

}

/\* Output:

{

    year: 2019

    month: 11

    day: 21

    hour: 10

    minute: 49

    second: 45

    uni: 1574329785

}

1574329785

2019-11-21 10:49:45

\*/

This functionality has to be used from the time module.

See $ 13.2 (guessing\_game.v) for time.now() and calc\_unix():

The now() function constructs a Time strict with the current data and time:

>>> time.now()

{

year: 2019

month: 11

day: 14

hour: 13

minute: 35

second: 13

uni: 1573734913

}

To get a console process to pause during a number of ms (say 500), use: time.sleep\_ms(500)

The function get\_fmt\_date\_str allows to format a tume instance according to a specified format.

In the following example instead of printing out data, we will use the assert function (see § 3.2)

Example see dates\_and\_times.v

You construct a datetime from the time.Time struct:

import time

fn main() {

  // make a Time instance for the datetime: '11.07.1980 21:23:42'

  t :=    time.Time{  year:     1980,

                      month:    7,

                      day:      11,

                      hour:     21,

                      minute:   23,

                      second:   42,

                      uni:      0 }

  println(t)

  assert time.is\_leap\_year(1600) == true

  assert  '11.07.1980 21:23:42' == t.get\_fmt\_str(.dot, .hhmmss24, .ddmmyyyy)

  assert  '11.07.1980 21:23' == t.get\_fmt\_str(.dot,.hhmm24,.ddmmyyyy)

  assert  '21:23' == t.get\_fmt\_time\_str(.hhmm24)

  assert  '9:23 p.m.' == t.get\_fmt\_time\_str(.hhmm12)

  assert  '21:23:42' == t.get\_fmt\_time\_str(.hhmmss24)

  assert  '1980-07-11' == t.get\_fmt\_date\_str(.hyphen, .yyyymmdd)

  assert  '11.07.1980' == t.get\_fmt\_date\_str(.dot,.ddmmyyyy)

  assert 'Jul 11' == t.get\_fmt\_date\_str(.space,.mmmd)

  assert  '21:23:42' == t.get\_fmt\_time\_str(.hhmmss24)

  assert  '21:23' == t.get\_fmt\_time\_str(.hhmm24)

  assert  '9:23:42 p.m.' == t.get\_fmt\_time\_str(.hhmmss12)

  assert  '9:23 p.m.' == t.get\_fmt\_time\_str(.hhmm12)

  assert  '11/07/80'   == t.get\_fmt\_date\_str(.slash,.ddmmyy)

}

## 4.9. Pointers References

?? should better come at a later stage, at least parts of it, likr in ch 9 structs

V, like most other low level (system) languages as C, C++ and D, has the concept of pointers.

Unlike Java and .NET, V gives the programmer control over which data structure is a pointer and which is not. By giving the programmer control over basic memory layout, V provides you with the ability to control the total size of a given collection of data structures, the number of allocations, and the memory access patterns, all of which are important for building systems that perform well: pointers are important for performance and indispensable if you want to do systems programming, close to the operating system and network.

Pointers in general should be rarely used in V.

But calculations with pointers (so called *pointer arithmetic,* e.g. pointer + 2, to go through the bytes of a string or the positions in an array), which often lead to erroneous memory access in C and thus fatal crashes of programs, are *not* allowed in V, making the language *memory-safe*. V pointers resemble more the *references* from languages like Java, C# and VB.NET .

By default you cannot calculate with pointer values in programs. (see pointer\_arithmetic.v)

?? However pointer arithmetic is allowed inside unsafe blocks; it's used a couple of times in vlib.

Because pointers are somewhat unknown to contemporary OO-programmers, we will explain them here and in the coming chapters in depth.

Programs store values in memory, and each memory block (or word) has an address, which is usually represented as a hexadecimal number, like 0x6b0820 or 0xf84001d7f0

V has the *address-of* operator **&** (‘ampersand’) , which, when placed before a variable, gives us the memory address of that variable. In other words: it creates a reference (which is a voidptr type) to a memory location

The following code-snippet for example defines a pointer ptr as the address of an integer i, and prints this address out (this value will be different every time you run the program):

Listing 4.15 - pointer.v

fn main() {

  i := 100

  ptr := &i

  println(ptr) // 0x7ffd82de0f5c

  m := \*ptr

println(m)

}

This address can be stored in a special data type called a *pointer.* In this case it is a pointer to an int, here i.

So ptr stores the memory address of i; it points to the location of i, it *references* the variable i.

V pointers (references ??) are similar to Go pointers and C++ references.

A pointer variable contains the memory address of another value: it points to that value in memory.

The size of a pointer is 4 bytes on 32 bit machines, and 8 bytes on 64 bit machines, regardless of the size of the value they point to.

Pointers can be declared to reference a value of any type, be it primitive or structured. Using a pointer to refer to a value is called *indirection.*

A pointer variable is often abbreviated as ptr.

The symbol **\*** can be placed before a pointer like \*ptr , and then it gives the value which the pointer is pointing to; it is called the *dereference* (or *contents* or *indirection) operator*; another way to say it is that the pointer is *flattened*.

We could represent the memory usage as: i

100

ptr

0x7ffd82de0f5c

Figure 4.4: Pointers and memory usage

Program string\_pointer.v gives us an example with strings.

It shows that assigning a new value to \*p changes the value of the variable itself (here a string).

Listing 4.22 – string\_pointer.v:

func main() {

  str := 'good bye'

  ptr := &str

  \*ptr = 'ciao'

println(ptr) // prints address

println(\*ptr) // prints string

println(str) // prints same string

}

Output:

0x2540820

ciao

ciao

By giving \*ptr another value, the ‘object’ (in this case the string str) is also changed.

Schematically in memory: str

ciao

0x2540820

Ptr

Figure 4.5: Pointers and memory usage, 2

Exercise: int\_pointer.v: What is the output of this code snippet ?

fn main() {

  y := 3

  mut x := &y

  \*x = 4

  println(y)

}

One advantage of pointers is that you can pass a reference to a variable (for example as a parameter to a function), instead of passing a copy of the variable. Pointers are cheap to pass, only 4 or 8 bytes. When the program has to work with variables which occupy a lot of memory, or many variables, or both, working with pointers can reduce memory usage and increase efficiency. Pointed variables also persist in memory, as long as there is at least 1 pointer pointing to them, so their lifetime is independent of the scope in which they were created.

On the other hand, because a pointer causes what is called an indirection (a shift in the processing to another address), prohibitive use of them could cause performance decrease.

Pointers can also point to other pointers, and this nesting can go arbitrarily deep, so you can have multiple levels of indirection. But in most cases this will not contribute to the clarity of your code, so it is better to avoid that.

In many cases V makes it easier for the programmer and will hide indirection like for example performing an automatic dereference.

(dereferencing.v also works, useful ??).

byteptr / voidptr

V has 2 basic pointer types: *byteptr* and *voidptr*

byteptr is like char\* or byte\* in C.

voidptr is like void\* in C, it can point to anywhere in memory, you just don't know the type of the thing it points to, you need to cast it to some other pointer type to dereference it.

= type holding a numeric value of a location in computer memory regardless its type or other properties, where user is responsible for the correct casting (converting) to certain types or manipulation with the value stored within the memory starting on that address - offsetting the stored memory address up to specified data container size (or its multiples). A void pointer *does not store the value’s type*.

Only to be used in unsafe code ?? example

A pointer that has value 0 is a null pointer. In pure V code, these cannot be created. But for example when interacting with C, you must be able to test whether you have a null pointer or not. This is done with the following function: if isnil(v voidptr) { // handle null pointer }

(For an example: see ch 9 rp\_game.v)

Exercise: Understand what happens in this code snippet (see pointer\_error.v)

fn main() {

  mut i := 7

   p := &i

   println(p) // 000000000062FE44

   i = p      // implicit dereference

   println(i) // 6487620

}