# Chapter 11 – Modules and Testing

## 11.1 Making and using modules

V is a very modular language. Creating reusable modules is encouraged and is very simple. Modules are called libraries or packages in other languages.

To create a new module, create a directory with your module's name and .v files with code:

cd ~/code/modules // wherever you want to create a module e.g. /home/user

mkdir mod1 // same name as the module name

Start editing the file: mod1/mod1.v

In mod1.v you code:

**module mod1**

**fn init() {}**

**pub** fn say\_hi() {

println('hello from module mod1!')

}

pub tells us that the function say\_hi() is exported: it can be used in another module, outside of the module mod1.

The init() function is optional: it can contain code that is executed when the module is imported in another program; it cannot be public (pub).

v.mod:

(?? Not obligatory)

The properties and dependencies of a module can be describes in a v.mod file. This is a JSON text file containing minimally:

#V Project#

Module {

  name: 'project1',

version: '0.0.1',

  description: 'description of project1',

  dependencies: []

}

A concrete example:

Module {

name: 'vpkg'

version: '0.5.3'

deps: ['nedpals.vargs']

}

If you want to create a binary mod1.o file, build it with v build module ~/code/modules/mod1 (you have to give the complete path to the module).

Header files: extension **.vh**

(??) This *creates a module header file* mod1.vh (temporarily), in the folder $HOME/.vmodules. This is removed, and then the binary file mod1.o is created in .vmodules/~/code/modules/mod1 in your home directory.

v uses header files for precompiled modules. They are very different from C header files, almost closer to a definition file: they contain the declarations and signatures of the module structs and functions. (?? Why are they automatically removed)

Here is an example: (see strings.vh)

// vlib/strings module header

module strings

struct Builder {

  mut :

  buf []byte

  pub :

  len int

}

fn repeat (c byte, n int) string

fn new\_builder (initial\_size int) Builder

fn (b mut Builder) write (s string)

fn (b mut Builder) writeln (s string)

fn (b mut Builder) str () string

fn (b mut Builder) free ()

fn levenshtein\_distance (a, b string) int

fn levenshtein\_distance\_percentage (a, b string) f32

fn dice\_coefficient (s1, s2 string) f32

(?? Nov 11:

Windows 10:

Building module "mod1" (dir="mod1")...

V error: mod1 doesn't exist)

Building module "E:.Vlang.The\_Way\_to\_V.Chapter\_11\_Modules\_and\_Testing.mod1" (dir="E:\Vlang\The\_Way\_to\_V\Chapter\_11\_Modules\_and\_Testing\mod1")...

Generating a V header file for module `E:\Vlang\The\_Way\_to\_V\Chapter\_11\_Modules\_and\_Testing\mod1`

V panic: failed to create file "**C:\Users\CVO\.vmodules**\E:\Vlang\The\_Way\_to\_V\Chapter\_11\_Modules\_and\_Testing\mod1.vh"

print\_backtrace\_skipping\_top\_frames is not implemented on this platform for now...)

Works on Linux: v build module /home/ivo/mod1

* Building module “mod1” (dir = “/home/ivo/mod1”)…

Generating a V header file for module “/home/ivo/mod1”

Building /home/ivo/**.vmodules**/home/ivo/**mod1.o**

That's it, you can now use it in your code (use\_mod1.v):

**import mod1**

fn main() {

**mod1.say\_hi()**

}

When a project uses facilities from other modules, it has to import them. For example: if it wants to use functionalities from module mod1, it can do so by importing that module: import mod1

You import a module, not a file. Modules can’t have global variables. Imported modules are compiled statically into a single executable, compilation is fast, so recompilation is not a problem.

This also works on Windows:

v using\_mod1.v // -> create an executable using\_mod1, compiling and linking in the code from module mod1

using\_mod1 // => 'hello from module mod1!

Test: if pub is removed from say\_hi(): panic: using\_mod1.v:4, function `mod1.say\_hi` is private

You can have as many .v files in mod1/ as you want, but the convention is to have one source file with a name identical to the module name. It is this file which should contain the init() function if that is needed. Other files in this module folder can have different names. All files that are part of the module must start with module mod1. Every source file belongs to one (and only one) module. All modules are compiled statically into a single executable.

Note that you have to specify the module name every time you call a public function from it, as in mod1.say\_hi(). This may seem verbose at first, but it makes code much more readable and easier to understand, since it's always clear which function from which module is being called, especially in large code bases.

Submodules and module alias with as:

You can also create submodules in subfolders. They are *independent modules*, which are just organized in a hierarchical fashion under the namespace of the containing module. You can import them using the dot notation like this:  
folder structure: module1/submodule2/ => import module1.submodule2 ,

example: import math.complex  
or  
folder structure: module1/submodule2/ => import module1.submodule2 as sm

If you don’t use an alias, you can call functions in your code as: submodule2.func1()

For readability, module names should be short (like the vlib modules `net`, `os`, `gg`, `gx`, etc), under 10 characters.

Here is an example taken from vlib: base64\_test.v contains the tests for module base64.

In order to test the base64 functionality, base64\_test.v needs to do an import encoding.base64; this imports the code from module base64, which is contained in vlib/encoding/base64.v

base64.v: module base64

// rest of the code

fn decode(data string) string { }

base64\_test.v: import encoding.base64

fn test\_decode() { }

Question: Give base64 the alias b64 and import it

Answer: import encoding.base64 as b64

Importing several modules:

The import of more than one module can also be abbreviated.

Instead of:

import os

import time

write:

import (

  os

  time

)

This is like the mandatory const notation.

Module main: Programs, unlike libraries, need fn main, an entry point. It must be defined in a module main. If your program has more than one source file, you need to use module main in each of the files. If your program has only one source file, you don’t need to use module main. We have also seen that in that case you can often leave out the main() function itself. (?? Example)

Name conflicts are eliminated:

If the same function name exists in two different modules with different code, then the module name acts like a namespace since you prefix the function name with the module name when you call it, as in: mod1.say\_hi() and mod2.say\_hi() . The two say\_hi() functions will not conflict with each other, because they exist in two different modules and the module name prefix disambiguates them to the compiler. That also works well for external libraries you download to use in your code.

*You must always specify the full path, collisions are impossible.*

Cyclic dependencies between modules:

Suppose that module a imports b and module b imports module c, and for some reason module c needed a function from module a and imports it. This cyclic dependency is not allowed by the V compiler.

Modules can import other modules, but circular imports are not allowed

The design of the application has to be changed or cleaned up to remove circular imports.

How does V search for a module on your system:

V searches your module in the following paths (it follows the same mechanism as Go):

1. <working directory>/<folder of the module>

2. ~/.vmodules/<folder of the module> (e.g.: C:\Users\CVO\.vmodules\nedpals\args)

3. <working directory>/modules/<folder of the module>

Using multiple modules:

Here is an example of using 2 modules, which alse demonstrates the use of the init() function:

Folder structure:

app.v

modulea/modulea.v

moduleb/moduleb.v

modulea/modulea.v:

module modulea

fn init() {

  println('Initializing module a!')

}

pub fn hello() {

print('Hello ')

}

moduleb/moduleb.v:

module moduleb

fn init() {

  println('Initializing module b!')

}

pub fn world() {

println('world!')

}

app.v:

import modulea

import moduleb

modulea.hello()

moduleb.world()

v run app.v ============ running app ============

Initializing module a!

Initializing module b!

Hello world!

Some remarks:

* Unused module errors are warnings in non-production builds and errors in production builds:

warning: using\_mod1.v:1:6: the following imports were never used:  \* modulea

* Modules are cached for faster compilation
* The V compiler is now itself a module that can be used by other programs.

Other examples of constructing modules:

§ 14.2 B. shows how a more realistic module is developed, using the Open Weather Map web service, the *owmw* module.

## 11.2 Visibility outside of a module

We now that inside a module, variables are by default immutable, and local to the functions in which they are defined. Struct fields are also private and immutable by default, making structs immutable as well. To export constants, types, structs and functions, we need the pub keyword.

pub fn public\_function() {

}

fn private\_function() {

}

Structs and constants: are only known in the module in which they are defined (so they are private to their parent module). So if you want to export (make public) a struct Example, you have to write: pub struct Example

Different combinations pub and mut (so called *access modifiers*) can change the access to struct fields. There are 5 possible options:

struct Example {

a int // immutable, read-only field, not visible outside this module (the parent module)

mut: // mutable inside this module only, not visible outside this module

b int

c int

pub: // visible outside this module (public or exported), but read-only

d int

pub mut: // visible outside this module, and mutable

e int

\_\_global: // public and mutable both outside and inside the parent module

f int

}

\_\_global: This is not the same as global variables; V doesn’t have them. This is only for struct fields.

This option is not recommended to use.

?? see *visibility.v* , but compilation error (Nov)

For example, here's how the string type is defined as a struct in the builtin module:

struct string {

str byteptr

pub:

len int

}

It's easy to see from this definition that string is an immutable type.The byte pointer with the string data is not accessible outside builtin at all.The len field is public, but read-only:

fn main() {

str := 'hello'

len := str.len

str.len++ // <- compile error:

}

This gives the error: cannot modify immutable field `len` (type `string`)

mutable\_struct.v shows the behaviour inside the parent module:

struct Point {

    x int

mut:

    y int

}

fn main() {

    mut point := Point{x:10, y:20}

    // point.x++ // won't compile: cannot modify immutable field `x` (type `Point`)

    point.y++ // OK

    println(point)

}

/\*

{

    x: 10

    y: 21

}

\*/

Examples of exported struct fields: ??

## 11.3 Testing a module

It is good practice to separate the test code from the real production code of an app or module, in fact V forces you to do so (see dnc\_sum\_test.v).

V separates the test files of a project in files that are named as **\*\_test.v**, for example: hello\_test.v

These test files contain the test functions, whose name starts with **test\_** , for example: test\_hello

Within these functions, assert is used to test the equality of expressions:

if assert returns true, the test succeeds; if it returns false, the test fails.

Example1: to test the function hello() in the following module hello, which lives in a file hello.v in the folder hello:

Listing 9.3A hello.v:

module hello

pub fn hello() string {

  return 'Hello world'

}

We write a test function test\_hello() in hello\_test.v, which sits in a folder at the same level as the subfolder hello. To have the hello() functionality, we need to import hello:

Listing 9.3B hello**\_test**.v:

import hello

fn **test\_**hello() {

**assert** hello.hello() == 'Hello world'

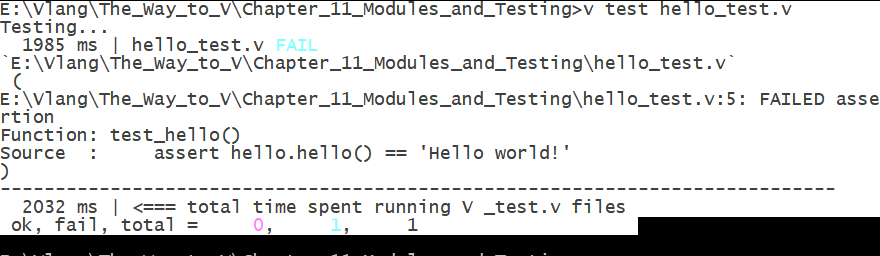
**assert** hello.hello() == 'Hello world!' // failing assertion

}

To run the tests do: v test hello\_test.v

Which gives as output:

(module hello and file hello\_test.v are compiled, and the test\_ functions are executed)



With the failed assert commented out, we get no output, so everything is ok!

Example2:

To add some tests, first code the following function in mod1.v:

pub fn say\_hi\_str() string {

  return 'hello from mod1!'

}

Then add the test code in mod1\_test.v:

import mod1

fn test\_say\_hi\_str() {

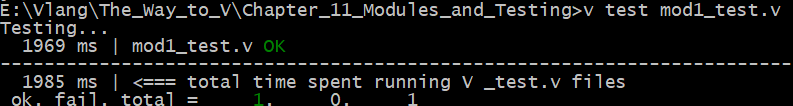
    assert mod1.say\_hi\_str() == 'hello from mod1!'

    // assert mod1.say\_hi\_str() == 'Hello world!'

}

To test an entire module mod1 do:  v test mod1\_test.v

If you uncomment the 2nd line, you get a failed assertion.



For a 3rd example: see *sum\_test.v* and *module sum*

Run the tests yourself for this example!

For running V’s built-in tests: see ch 2 – § 2.2 Testing the installation

## 11.4 Building a module for the game of life

In § 7.1.2 a V example program was shown, that implemented the game of life in some 80 lines. In this second iteration (see game\_of\_life/life.v) it is shown that the bulk of the functionality is captured in a separate module, called *automaton*. In this way the main program only takes less than 25 lines:

import (

  automaton

  time

)

fn print\_automaton(a &automaton.Automaton) { … }

mut aut := automaton.gun() // initialization

for {                      // game-loop

    aut.update()           // calculate new generation

    print\_automaton(aut)   // display new generation

    time.sleep\_ms(100)

}

In the module’s code (*automaton/automaton.v*) the game logic is abstract in a struct Automaton:

pub struct Automaton {

pub mut:

  field &A2D

  new\_field &A2D

}

Where A2D is another struct defined as:

pub struct A2D {

pub mut:

  maxx int

  maxy int

  data &int

}

This is a more primitive data structure. It manipulates a pointer to an integer, which is code which must be put inside an unsafe block.

It also defines inlined functions (see ch 12) with [inline], to improve performance.

The data field of an A2D struct is initialized like this:

data: &int( calloc( sizeof(int) \* maxy \* maxx )

*calloc* is a V function defined in builtin.v

It calls the C library function void \**calloc*(size\_t nitems, size\_t size), with its second argument set to 1.  This function allocates the requested memory and returns a pointer to it. The difference between malloc and calloc is that malloc does not set the memory to zero, whereas calloc sets allocated memory to zero.

*sizeof* takes the memory size of an int, which is 4 bytes (int32).

## 11.5 Package managers and installing modules

To provide for a flourishing ecosystem of modules, V has an easy to use and centralized package manage, which is called vpm

vpm is a program which can be found in [path\_to\_v]\tools (for example: c:\v\tools\vpm.exe)

The repository can be found at: <https://vpm.best/> (same as <http://vpm.vlang.io./> ).

At the time of writing (Dec 2019), this contains 42 modules.



A module has a 2-part name: creator.modname, for example: *nedpals.args*

Clicking on a module leads you its GitHub repo (<https://github.com/nedpals/vargs>)

v package management commands:

First compile vpm with: c:\v\tools> v –prod vpm.v

Then vpm shows:

Usage:

b) v search keyword1 [keyword2] [...]

c) v install module [module] [module] [...]

d) v update [module] [...]

e) v remove [module] [...]

You can also pass -h or --help after each vpm command from the above, to see more details about it.

Here is some more explanation:

?? search keywords Search the https://vpm.vlang.io/ module repository for matching modules and shows their details.

**install** <module> Install a user module from https://vpm.vlang.io/.

?? update [module] Updates an already installed module, or ALL installed modules at once, when no module name is given.

?? remove [module] Removes an installed module, or ALL installed modules at once, when no module name is given.

Installing a module:

This can be done directly from the v command line.

Installing modules is as easy as: v install creator.modname

This installs the source code of the module in folder .vmodules in your home directory, ready for use in your own app.

A module modname is installed by downloading its source code to a $HOME/.vmodules**/**modname

If it is a submodule mod1/mod2, its source code is stored in $HOME/.vmodules/mod1/mod2

Examples:

c:\v>**v install nedpals.args**

Installing module "nedpals.args" from https://github.com/nedpals/v-args to C:\Users\CVO\.vmodules/nedpals/args ...

🡪 in folder .vmodules the source code from the module is stored in subfolder nedpals/args

c:\v>v **install regex**

On Windows:

Installing module "regex" from https://github.com/ShellBear/v-regex to C:\Users\CVO\.vmodules/regex ...

🡪 in folder .vmodules the source code from the module is stored in subfolder regex

On Linux:

Installing module "regex" from https://github.com/ShellBear/v-regex to /home/ivo/.vmodules/regex ...

🡪 in folder .vmodules the source code from the module is stored in subfolder regex

Submitting your V module:

This takes a couple of seconds. After authorizing vpm to access your GitHub repo, you only have to fill in this form:



Vpkg: (?? Still used)

Vpkg is an alternative package registry for managing and publishing your V modules, currently (Nov 2019) containing 43 modules: <https://vpkg-project.github.io/registry/>

This site contains the installation instructions: <https://github.com/vpkg-project/vpkg>

To install a module, give the command: vpkg get v-bitmap

Attempt to install:

On Windows: (Dec 3): failed

On Linux: idem

install.v: syntax error: unexpected `import`, expecting `module` , at import api

Other repositories:

* <https://vlang.io/#software>
* The awesome-v site: <https://github.com/vlang/awesome-v>

For concrete examples of using one or more external modules in your own app, see chapter 14:

* 14.2 OWMW: an API wrapper for Open Weather Map, §14.2 C

We also give an overview of some interesting external modules and how to use them.