# Chapter 12 – Advanced topics

## 12.1 Generic types and functions

?? V will include generics as of ….

When a struct or a function is generic it has <T> written after its name, like in:

struct Repo<T>

fn (r Repo<T>) find\_by\_id(id int) ?T

V has the concept of *generics*, which means for example defining 1 function which can be applied to a number of variable types (see § 13.1).

Their type signature is:

fn fun1<T>(value T) T {}

which reads as : fun1 is a generic function <> with generic type T. It takes an argument value of that type, and has a return value of that type.

Generic functions can also have varargs support.

V generates executable code for every concrete type a generic function is used with.

Example: generics0.v

In this program, plus takes 2 arguments of a generic type T, and returns a value of that type. It is called with integer, float and string arguments:

fn plus<T>(a, b T) T {

    return a + b

}

fn main() {

    println(plus(10,11)) // 21

    println(plus(1.0, 2.0)) // 3.000000

    println(plus('hello', ' world!')) // hello world!

}

?? does not compile yet : Preliminary example: Listing 12.1 generics.v:

## ?? 12.2 Lambda functions

## 12.3 Attributes

Syntax: between [ ]

Example:

[json]

struct User {

name string

age int

is\_registered bool

}

🡨 this makes the compiler generate json.encode and json.decode (??)

Kinds of attributes:

[json]

[raw] Raw json fields via the [raw] attribute

[json: ] to indicate custom JSON field names

[live] see $ 3.11

[deprecated] attribute.

[if] function attributes for compile time function exclusion for performance.

[typedef] attribute for imported C struct typedefs.

[inline] Functions can now be inlined via this attribute:

Inlining means that the [compiler](https://en.wikipedia.org/wiki/Compiler) substitutes the body of the function inline by performing [inline expansion](https://en.wikipedia.org/wiki/Inline_expansion), i.e. by inserting the function code at the address of each function call. This removes the overhead of a function call, which can substantially improve performance if functions are called many times.

Examples: see ch 11 – game\_of\_life/automaton/automaton.v

[inline] pub fn (a &A2D) get(x, y int) int { … }

[inline] pub fn (a &A2D) set(x, y int, newval int) { … }

[inline] pub fn (a &A2D) clear() { … }

- @ for escaping keywords (e.g. struct Foo { @type string }).

## 12.4 Compile-time function execution and code generation

(Apr 5 2019) V already supports compile-time function execution: examples ??

$if (see ch 3)

Having built-in JSON support is nice, but V also allows you to create efficient serializers for anything using generic functions:

?? doesn’t compile:

fn decode<T>(data string) T {

mut result := T{}

for field in T.fields {

if field.typ == 'string' {

result.$field = get\_string(data, field.name)

} else if field.typ == 'int' {

result.$field = get\_int(data, field.name)

}

}

return result

}

fn decode\_User(data string) User {

mut result := User{}

result.name = get\_string(data, 'name')

result.age = get\_int(data, 'age')

return result

}

- Comptime codegen (foo.$method() where method is a string).

## 12.5 V script and vsh - Cross-platform shell scripting

Scripting is not necessary for building an executable or a library.

V implements a simpler and cross-platform alternative to Bash script tooling.

All V deployment scripts were rewritten from Bash to V (for example: the scripts for V playground, V binaries for early access).

V can be used as an alternative to Bash to write deployment scripts, build scripts, etc.

*The advantage of using V for this is the simplicity and predictability of the language, and cross-platform support: V scripts run on Unix-like systems as well as on Windows.*

Examples of usage:

* creating installers
* running external programs like qemu
* packaging code into .iso's to boot on a system
* publishing
* building multiple executables/libraries
* or whatever other tasks that need to be done after building v code.

V scripts have the extension **.vsh**

That way, all functions in the os module are global, so that you can use ls() instead of os.ls(), for example.

On Unix systems you have #! support for executing your V scripts (example ??)

?? Example:

Listing 14.1 example\_script.vsh - Doesn’t compile on Windows /

Listing 14.2 deploy.vsh - Doesn’t compile:

rm('build/\*')

// Same as:

for file in ls('build/') {

  rm(file)

}

mv('\*.v', 'build/')

// Same as:

for file in ls('.') {

  if file.ends\_with('.v') {

    mv(file, 'build/')

  }

}

Now you can either compile this like a normal V program and get an executable you can deploy and run anywhere: **v deploy.vsh && ./deploy**

Or just run it more like a traditional Bash script:**v run deploy.vsh**

## (??) 12.6 ORM

V has a built-in ORM that supports Postgres, and will soon support MySQL and SQLite.

The benefits of V ORM:

 One syntax for all SQL dialects. Migrating to a different database becomes much easier.

 Queries are constructed with V syntax. There's no need to learn another syntax.

 Safety. It's impossible to construct a SQL query with an injection.

 Compile time checks. No more typos that can only be caught at runtime.

 Readability and simplicity. You don't need to manually parse the results and construct objects.

struct Customer {

id int

name string

nr\_orders int

country string

}

db := pg.connect(db\_name, db\_user)

nr\_customers := db.select count from Customer

println('number of all customers: $nr\_customers')

uk\_customers := db.select from Customer where country == 'uk' && nr\_orders > 0

println(uk\_customers.len)

for customer in uk\_customers {

println('$customer.id - $customer.name')

}

customer := db.select from Customer where id == 1 limit 1

println('$customer.id - $customer.name')

new\_customer := Customer{name: 'Bob', nr\_orders: 10}

db.insert(new\_customer)

## 12.7 Translating C/C++ to V

?? When available ?? [wip, will be available in V 0.3]

An entire C project or codebase can be translated to V, offering you the safety and simplicity of V, and making your build times up to 200 x faster. This is done through an automatic translator that supports the latest C standards. Simple C++ projects can now (Apr 2019) also be translated, a more complete support for translation of all C++ standards in under way.

C to V translator is for translating C code into easy to digest and easy to maintain V code. Another useful thing in this translator is that you can port any C library easily instead of having C bindings which other languages do.

Also if one line of V ever gets to the place where it can do the same work as two lines of C, then you only have half a many lines of code to maintain. The same way that some languages on Rosetta Code can do the same code problem in half the lines of code as other languages and at the same time be more readable.

The Clang parser is used for translating C/C++ to V.

For example: DOOM and DOOM 3 were translated to V (see <https://github.com/vlang/doom> ), as well as [LevelDB](http://vlang.io/) and [SQLite](http://vlang.io/).

(Nov 29: both on Windows and Linux: errors in cmake . , failed to make chocolate-doom)

Here is an example of a translation:

**C++** (test.cpp):

#include <vector>

#include <string>

#include <iostream>

int main()

{

std::vector<std::string> s;

s.push\_back("V is ");

s.push\_back("awesome");

std::cout << s.size() << std::endl;

return 0;

}

**V** (test.v)

fn main() {

mut s := []string

s << 'V is '

s << 'awesome'

println(s.len) // 2

}

The translation from test.cpp to test.v is done through the command:

To translate C to V: v translate file.c

Or: v translate test.cpp

which generates the file test.v

When should you translate C code and when should you simply call C code from V?

If you have well-written, well-tested C code, then of course you can always simply call this C code from V.

It is possible to translate files one by one and gradually replace C with V. Object files compiled with V have full binary compatibility with object files produced by C, and can be a drop-in replacement.

Translating it to V gives you several advantages:

- If you plan to develop that code base, you now have everything in one higher-level language V, which is much safer and easier to develop in than C.

- Outputting assembly is much faster, so testing and iterative development is faster. Outputting to C allows intensive advanced optimisation when desired.

- Cross-compilation becomes a lot easier. You don't have to worry about it at all.

- No more build flags and include files either.

## 12.8 Calling C and C++.

(??!!) A V program doesn’t need libc. >< stdlib

V has a libc dependency (May 3), everything other than libc is linked statically, so the binaries do not depend on other external libraries.

### 12.8.1. Interacting with C

V can seamlessly interact with C: You can reuse all your existing C code and libraries and call them from V without any performance costs. The ability to call C from V solves most of the ecosystem issues because everything has already been done in C.

You can call any C function abc like this: C.abc( )

#include 🡨 to include C header files

Flags:

#flag 🡨 to link with object files

Add #flag directives to the top of your V files to provide C compilation flags like -l (small letter l) for linking, -I (capital letter I) for adding include files locations, -D for setting compile time variables, etc.

You can use different flags for different targets. Right now, linux , darwin , and windows are supported.

For now you have to use one flag per line:

#flag linux -lsdl2

#flag linux -Ivig

#flag linux -DCIMGUI\_DEFINE\_ENUMS\_AND\_STRUCTS=1

#flag linux -DIMGUI\_DISABLE\_OBSOLETE\_FUNCTIONS=1

#flag linux -DIMGUI\_IMPL\_API=

To include one or a number of big C files, you can decrese compilation time by first building the C to .o files, and then including them in your V code with a #flag like this: #flag @VROOT/thirdparty/mylib/myfile.o

Linking to an external dependency:

If you need to link to an external dependency glfw3, you do it with the #flag syntax:

module mod1

#flag linux -lglfw3

Examples of V code using C: *call\_c.v:*

#include <stdlib.h>

fn main() {

C.puts('hello, world from V!')

}

🡨 (Nov 1: no output on Windows)

'hello'.cstr()

[1,2,3].carray()

In general (Apr 3 ’19):

#flag linux -lmylib  
#include <mylib.h>

C.mylib\_fn()

Here is an example from the Docs which connects with and gets info from a sqlite database:

Listing 3.1 – sqlite3.v:

(?? Not yet tested)

#flag -lsqlite3

#include "sqlite3.h"

struct C.sqlite3

struct C.sqlite3\_stmt

fn C.sqlite3\_column\_int(C.sqlite\_stmt, int) int

fn main() {

path := 'sqlite3\_users.db'

db := &C.sqlite3{}

C.sqlite3\_open(path.cstr(), &db)

query := 'select count(\*) from users'

stmt := &C.sqlite3\_stmt{}

C.sqlite3\_prepare\_v2(db, query.cstr(), - 1, &stmt, 0)

C.sqlite3\_step(stmt)

nr\_users := C.sqlite3\_column\_int(res, 0)

C.sqlite3\_finalize(res)

println(nr\_users)

}

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

V uses voidptr for C's void\* and byteptr for C's byte\* or char\*.

To cast voidptr to V references use user := &User(user\_void\_ptr).

voidptr can also be dereferenced to V structs by casting: user := User(user\_void\_ptr).

Check out socket.v for an example of calling C code from V: <https://github.com/vlang/v/blob/master/vlib/net/socket.v>

To debug issues with the C code, v -show\_c\_cmd . is useful. It prints the C command that is used to build the program.

If you are calling C functions and need to access their error codes, you can do this with C.errno

V uses C’s atomic.

Detecting null:

C of course has null pointers. There is no null in V, but you can check if the pointer is 0 Theres an isnil method in builtin/builtin.v that does this:

if you receive a variable buf of type voidptr or char\* from C, you can check it in V via:

if isnil(buf) {

}

Call V from a C or C++ program: (i.e. DLLs)

I'll document that module\_\_fnname()

### 12.8.2. Interacting with C++

Mar 10 ’19: To use C++ from V, you’d have to either write a wrapper in C or translate C++ to V.

Calling V code is possible in any language that has C interop.

## ?? 12.8B unsafe code blocks

In an unsafe code block you can put code that violates some of V’s code safety restrictions, such as:

* Multi-level pointers in unsafe code (`\*\*\*\*int`)
* Working with pointers to integers

Here is an example (see unsafe1.v):

fn main() {

  a := 10

**unsafe** {

    println(a) // 10

  }

}

Here the unsafe is of course not needed, it only shows that it can be used.

A more concrete example can be found in § 11.4, where the code in the automaton module for a game of life program makes use of pointers to integers, which must be placed inside an unsafe block, for example:

[inline] pub fn (a &A2D) get(x, y int) int {

**unsafe** {

    mut e := &int(0)

    e = a.data + y\*a.maxx + x

    return \*e

  }

}

## ?? 12.9 Support for inline assembly

The V compiler emits machine code directly without generating assembly, but it can also emit C code, which can include inline assembly: *asm.v*

Assembly needs to be run inside an unsafe:

fn main() {

  a := 10

  unsafe {

    asm {

      mov eax, [a]

      add eax, 10

      mov [a], eax

    }

  }

}

## 12.10 Hot code reloading

(see folder hot\_reload)

Any time code is changed (and saved!) in a function marked with [live], that code is recompiled and re-integrated in the shared library.

(this means compile errors can occur while changing the code!)

To enable hot code reloading, build the source file with: v –live hot\_code.v

Which outputs a message like: compiling shared library took 1859 ms

If you run the program like any other as: v run hot\_code.v, you get the message:

INFO: run `v -live program.v` if you want to use [live] functions

Every function which must be able to reload must have the attribute [live]

Example 1: *message.v* (works on Windows / Linux)

import (

os

time

)

[live]

fn print\_message() {

  println('Hello! Modify this message while the program is running.')

}

fn main() {

  for {

    print\_message()

    time.sleep\_ms(500)

  }

}

Compile it with: v -live message.v

And run it as normal: message

Test it by changing the message in println, and see how the output changes!

It's not possible to modify types while the program is running.

Example 2: *bounce.v* (works on Linux)

Bounce.v: change rgb(0,0,0) in draw function or to other color.

Dec 11: does NOT work on Windows, window pops up, but crashes

v bounce.v and bounce do work!

Example 3: *graph.v* (works on Linux)

Change function y = …

Dec 11: does NOT work on Windows, window pops up, but crashes

v graph.v and graph do work!

## 12.11 JavaScript backend

Sep 15: V now has an early stage JavaScript backend:

*hi.v:*

fn main() {

for i := 0; i < 3; i++ {

println('Hello from V.js')

}

}

v -o hi.js hi.v && node hi.js

Hello from V.js

Hello from V.js

Hello from V.js

Dec 12 –

Doesn’t work on Windows:

V.js compiler not found, building...

c:\v\vlib\strings\builder\_js.v:46:27: `b` is declared as mutable, but it was never changed

44| }

45|

46| pub fn (b mut Builder) free() {

^

47| }

Failed.

After removing mut:

type `compiler.Parser` has no field or method `gen\_handle\_question\_suffix`

370| if p.tok == .question {

371| // `files := os.ls('.')?`

372| return p.gen\_handle\_question\_suffix(f, fn\_call\_ph)

^

373| }

374| else if !p.is\_var\_decl && is\_or\_else {

Failed.

On Linux: Same error

## 12.12 Concurrency and Data races

The concurrency model is very similar to Go. To run foo() concurrently, just call it with go foo(). Right now, it launches the function in a new system thread. Soon coroutines and the scheduler will be implemented.

**go** fn() {

…

}**() // 🡨 here fn is called**

|  |
| --- |
|  |

Every go call starts the call to function fn() in a new thread.

Every critical, mutually exclusive section must be contained within a lock block:

**lock** {

…

}

The method **sync.wait()** on a WaitGroup allows the main thread to wait until all coroutines have finished.

Here is an example of code that sends a value to another thread:

import (

  time

)

fn send(s string) {

  println(s)  // Alan

}

str := 'Alan'

go send(str)

// Make sure the non-main coroutine had the chance to finish.

time.sleep(1)

Examples: hot\_reload/bounce.v

(??) concurrent\_news\_fetcher.v