# Chapter 2 – Installation and the compiler environment

## 2.1 Architectures and compilers

It is commonly known that C++ and also Rust compile very slowly. One of the key goals of Go was fast compilation, and V compiles even faster than Go.

### Target OS’s and platforms:

V can compile (through C) to the following operating systems (OS):

* macOS (also named Darwin)
* Windows (7, 8 and 10)
* Linux
* \*BSD (FreeBSD, OpenBSD, NetBSD, DragonFlyBSD)
* Arch Linux
* Solaris
* Android
* Haiku OS
* Through musl support ( <https://en.wikipedia.org/wiki/Musl> ): V can now run on, for example, Alpine Linux.

It has also been compiled to an ARM device, namely a Raspberry pi 3b.

The list is steadily growing.

Right now only 64 bit (*x64/MachO architecture)* is supported fordirect generation of binary code (mode a).

(??) The compiler will be fully *parallellized* in the future, and so can use every processor in the development machine.

The compiler is named v.

### There are 2 kinds of compiler :

a) **V => C => machine code**

This is the primary, standard compiler, used for development and production.

<https://github.com/vlang/v/wiki/On-the-benefits-of-using-C-as-a-language-backend>

When you are ready to put your project in production, you need a fully optimized version of your code. V provides for this with a compiler that transpiles to human-readable C, which is then compiled to machine code with a C compiler (GCC, Clang, tcc or MSVC on Windows). In this way V profits from the sophisticated optimizations in these compilers, and it can be compiled to run on any platform: compile with v –prod to get optimized binary code.

Such builds are compiled ≈10 slower (~150k loc/CPU core) than compiling V directly to binary code, but they are still much faster than C++ builds.

b) **V => machine code:** see § 2.5

As the name says, this version is meant to be used during the development cycle. It is a light, simple, basic and non-optimizing compiler at the moment (optimization will be done in the future) and it is to be used for *fast debug development builds*.

How does it work? The parser is handwritten. It generates machine code (binary code) as it goes, in a single pass. No AST (Abstract Syntax Tree), no IR (intermediate code) or no assembly is generated; the ELF / Mach-o / PE binary structure is generated by itself. That's why it's so fast.

?? This is changed with the AST parser.

How fast is it? V compiles about *1 to 1.5 million lines of code (loc) per second per CPU core*, on an 8-core desktop machine compilation speeds are about 10 million lines per second.

This is

* + - 100-200 x faster than Jai compilation
    - 5000 x faster than C++ compilation

Compilation speed is linear, so compiling 100 million lines of code will always take only about 10 seconds.

Here is an example of compiling a V-version of Doom3:

cd doom3/

wc -l doom3.v # 458 713

time v doom3.v # 0.5s

Source code: <https://github.com/vlang/doom>

Comparison of compilation speed with other languages:

See <https://vlang.io/compilation_speed>

?? V will have its own linker written in V. (Apr 11).

In what language the compiler is written? The V compiler is self-hosting: the compiler written in V compiles itself with a previous V version.

The compiler was first bootstrapped with Go, then written in C (v.c), but it is now written in V (v.v).

The compiler size is about 600 KB, with zero dependencies.

It contains a simple builds- and dependency management system: no make files, compiler flags, include files are needed; No more build environments, makefiles, headers, virtual environments, etc.

All you have to run is the v command to build your project, no matter how big it is.

You get a single statically linked binary that is guaranteed to work on all operating systems (provided you cross compile) without any dependencies.

Downloading the C compiler: v.c (see C compiler source, for Windows: it is v\_win.c)

This can be done with: git clone --depth 1 --quiet https://github.com/vlang/vc

To download an old version version of the C compiler:

curl -O https://raw.githubusercontent.com/vlang/vc/master/v.c

### V also supports hot (or live) code reloading (see § 3.11):

The non-optimized development compilation could be a problem for industries where optimization is required during development (for example AAA games). In this case *code reloading* is supported with the #live directive. This means you can change code in your editor, and the resulting change in the running program can be seen immediately: you don't need to exit your game and recompile it! This saves precious development time, which makes it perfect for developing scientific applications and games. For an example in action, see <https://volt-app.com/img/lang.webm>

Mar 9 2019: Right now it uses simple dlopen behind the scenes. In the future I’m planning to modify the program directly as it’s running.

V also supports incremental builds (not yet, not needed ??):

In most cases you won't be building the entire project. V has a smart incremental build system, so compiling your changes will only take a couple of milliseconds. This means that only the latest source changes are recompiled, taking almost no time.

V does not use LLVM (why?? To avoid the dependency on the LLVM tool chain, and the larger executables)

The compiler changes the source files by formatting them with the code formatting tool **vfmt** (not yet, see § ??)

(??) Compiling Go to V: It will be possible to convert Go code to V, once coroutines and channels have been implemented. This could mean that V can be used as an alternative fast compiler for golang.

In the following sections we discuss the installation of V on the operating systems where it is feature complete, that is: Linux, Mac OS and Windows.

If you just want an easy install of V and its tools to start programming immediately download the binary release file for your OS from <https://vlang.io/>.

If you want to compile the code of the V system yourself: follow the instructions in section § 2.2.

The development of the V compiler follows a continuous integration (CI) cycle: CI has been set up (Travis + Azure). On every commit and PR it is made sure that V can compile itself, all tests pass, and all examples compile.

The different releases are kept here: <https://github.com/vlang/v/releases>

## 2.2 Installing the V compiler from source

### Installing a C compiler

You'll need Clang or GCC or Visual Studio. If you are doing development, you most likely already have one of those installed.

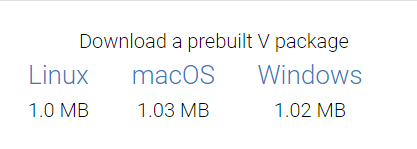
Otherwise, follow these instructions:

<https://github.com/vlang/v/wiki/Installing-a-C-compiler-on-Linux-macOS>

https://[github.com/vlang/v/wiki/Installing-a-C-compiler-on-Windows](https://github.com/vlang/v/wiki/Installing-a-C-compiler-on-Windows)

### Download V binaries

The V binaries are available from the website: <https://vlang.io/>



If you prefer to work at the command-line, here are the steps for Linux:

wget https://github.com/vlang/v/releases/latest/download/v\_linux.zip

unzip v\_linux.zip

cd v

sudo ./v symlink

The last line makes V globally available on your system.

On macOS use v\_macos.zip, on Windows: v\_windows.zip.

For any other OS, build V from source as in the next section.

### Build V from source

If you want to see the whole compilation process for yourself, this is the way to proceed:

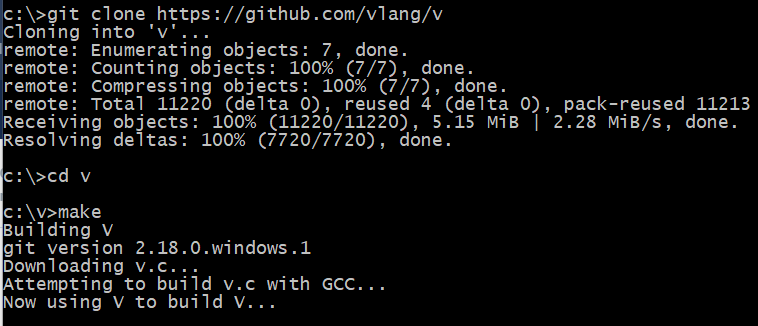
See <https://github.com/vlang/v#installing-v-from-source>

Open a terminal in a folder of your choice, e.g. *v-repo*:

git clone https://github.com/vlang/v

cd v

make



Now you will have a v compiler named v in folder v-repo.

The installation process is platform independent.

For Windows the make.bat file contains:

gcc -std=c99 -w -o v2.exe vc\v\_win.c (1)

v2.exe -o v3.exe v.v (2)

v3.exe -o v.exe -prod v.v (3)

We see the bootstrapping at work here :

In line (1) the C compiler for V (v\_win.c) is compiled with gcc to v2.exe

In line (2) this executable v2.exe is used to compile the V compiler for V to v3.exe

In line (3) this executable v3.exe is used to compile the V compiler for V to v.exe with production optimizations

Equivalently, you could do:

v -prod -o v v.v

to get a faster V.

For Linux, there is a Makefile.

Remark: If you get an error, try:

git pull

make

### Making V system-wide available:

To do that, the folder (let’s say this is called c:\v) where the v executable lives should be available in the PATH system environment variable.

In Windows and macOS this is taken care of by the installation process (any open Terminal sessions must be restarted for the changes to take effect).

Manual procedure for Windows:

* open Explorer, right-click This PC, Properties 🡪 the System settings window appears
* click Advanced system settings, then button Environment variables
* choose System Variables, Path, click button New, add c:\v
* click OK on all open windows

In Linux you must do this manually:

If the v compiler is installed in /usr/local/bin/v, add this line to your /etc/profile (for a system-wide installation) or $HOME/.profile:

export PATH=$PATH:/usr/local/bin/v

Or if the V compiler lives in $HOME/v

export PATH=$PATH:$HOME/v

Or you can create a symbolic link to the installation. Open a terminal in the installation folder and give the command:

sudo ln -s [path to v repo]/compiler/v /usr/local/bin/v

or shorter:

sudo ./v symlink

Remark: if you get a bash error: ./v permission denied

solution: chmod u+x v

### Installing through a Docker file:

Install Docker, then use the following commands:

git clone https://github.com/vlang/v

cd v

docker build -t vlang .

docker run --rm -it vlang:latest

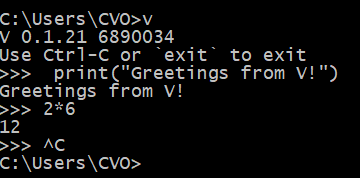
v

### Testing the installation

The command v -v shows you the version of your V compiler:

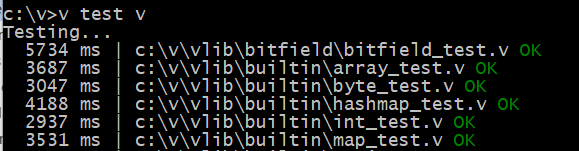


Running v in a terminal should open up the REPL window (see § 3.5):

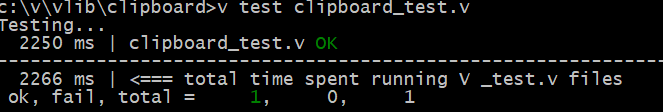


Also make sure V can compile itself: v v.v

Test your V environment by executing v test v to run all built-in tests and build all available examples.

‌

To run 1 particular testfile do: v test clipboard\_test.v

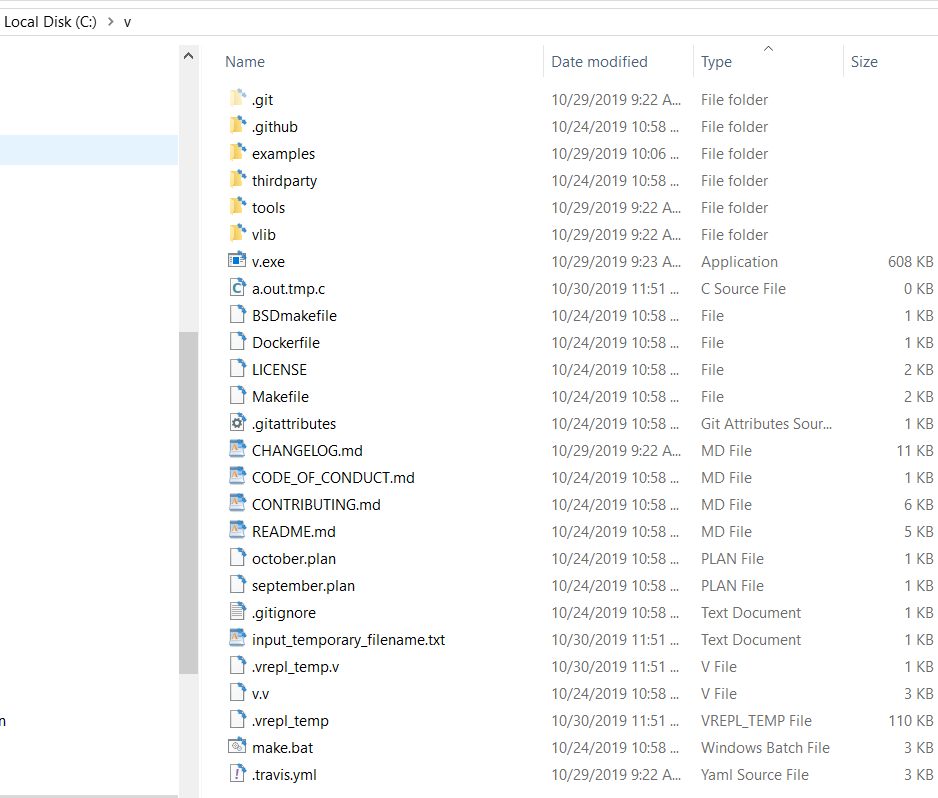


Tip: Try out some more built-in tests and look at the code!

Run v -h to see all the other options for the compiler.

### Directory structure of the V installation:

Here is a screenshot from an installation on Windows:



v.exe is the v compiler; vlib contains all the v sources (compiler and standard library)

The entire V language and its standard library are about 6-8Mb, which can be built in 0.3 seconds.

<https://github.com/vlang/v/wiki/The-V-language-now-compiles-itself-in-0.09-seconds>

Compare this with the compilation of the compilers of other languages:

|  |  |  |
| --- | --- | --- |
|  | **Space required** | **Build time** |
| Go | 525 MB | 1m 33s |
| Rust | 30 GB | 45m |
| gcc | 8 GB | 50m |
| Clang | 15-20 GB | 25m |
| Swift | 70 GB [\*](https://github.com/apple/swift#getting-started) | 90m |
| V | 8 MB | 0.3s |

### Updating the installation

V is evolving very fast: to get the latest version, issue the following command in the folder where V was installed: v up

If this procedure gives a problem, delete the v folder and repeat the procedure from: Compiling V from source.

If you have changed files from the installation:

git status: will give you what files you have changed that prevent the pulling

if you are sure that you do not want to keep your changes:

git checkout .(bewerkt)

git checkout master(bewerkt)

and then v up would be able to pull

## 2.3 Compiling and executing a simple program:

If your program’s source file is called program.v, then you can compile it with the command:

v program.v

This produces an executable, program.exe on Windows, or program on Linux-macOS, which you can start respectively as: program on Windows or ./program on Linux-macOS

This is a fast compilation, used in the development cycle. If you want a version optimized for production deployment, use the command: v –prod program.v

This takes somewhat longer to compile.

Let’s make this concrete. Using your favorite text-editor, make a file with the following code, and save this as hello\_world.v:

Listing 2.1 – hello\_world.v:

fn main() {

println('Hello, World from V!')

}

main() is most of the time not needed anymore in 1 file programs:

println('Hello, World from V!')

Compiling a program without imports also compiles ["builtin", "strings", "strconv.ftoa", "hash.wyhash", "strconv", "math", "math.bits"] (see v –verbose)

After your struct and function definitions, you can start defining variables without using main().

If you leave out main(), all functions have to be defined before the main code starts (see ch 6 – greeting.v)

Compile this first V-program as follows: v hello\_world.v

producing an executable with file size about 113 Kb on Windows for debug build, and 67K for a production build (v – prod). (On Linux 60K normal build, production build )

Then run it as hello\_world or ./hello\_world to produce the output: Hello, World from V!

Or combine the commands as: ./v hello\_world.v && ./hello\_world

This program prints the string Hello, World from V! to the terminal output through the println function.

Strings in V are enclosed in single quotes ' ' (preferred), but you can also use double quotes.

Compared to Go, we don’t need a package name or to import the fmt package, the function println is a core function known by the compiler.

V uses the **TMPDIR** environment variable, and will put .tmp.c files in TMPDIR/v/ . If you have not set it, a suitable platform specific folder (like /tmp) will be used.

On Windows:

A temporary C translation file *file.tmp.c* is stored in *C:\Users\CVO\AppData\Local\Temp\v*

Also the C output of the REPL is stored here in *.vrepl\_temp.tmp.c*

On Linux:

Here the temporary C files are stored in /tmp/v.

If the C compilation fails, the error message and where to find it in the C source is printed out.

If everything is ok, the C file is deleted and the program is run.

Compiling a directory:

v . compiles all v files in the current directory.

When given a directory as in v dir, all .v files contained in it will be compiled as part of a single main module (see ch 11). By default the executable will have the same name as the directory.

### Building Tetris and graphics programs:

In order to build Tetris and anything else using the graphics module, you will need to install glfw and freetype libraries.

**macOS:** brew install glfw freetype openssl

**Debian/Ubuntu:** sudo apt install libglfw3 libglfw3-dev libfreetype6-dev libssl-dev

**Arch/Manjaro:** sudo pacman -S glfw-x11 freetype2

**Fedora**: sudo dnf install glfw glfw-devel freetype-devel

**Windows**: git clone --depth=1 https://github.com/ubawurinna/freetype-windows-binaries [path to v repo]/thirdparty/freetype/

(the glfw dependency will be removed soon; Once C to V translator is more stable, I’ll translate all C dependencies, and it won't be necessary to install anything at all).

Solution on Linux: <https://stackoverflow.com/questions/17768008/how-to-build-install-glfw-3-and-use-it-in-a-linux-project>

See also document: How to compile tetris

Compiler options: (?? see compiler/main.v from line 1350 onwards)

v - h or v help or v help --verbose

produces a help text detailing all commands and options:

Usage: v [options/commands] [file.v | directory]

**General options:**

- To get current V version and display git hash of the compiler source: v -v or v version

- To symlinks the current V executable to /usr/local/bin/v, so that V is globally available (useful on unix systems): ./v symlink

-To execute all tests and build all example programs: v test v

- To start the REPL: v or v runrepl

**Compiling files:**

-To build a V program: v file.v resulting in file.exe (Windows) or ./file on \*nix systems.

- To build an optimized executable (O2): v -prod file.v

- To specify the executable’s name as program use: v -o program file.v

-o <file>.c Produce C source without compiling it.

-o <file>.js Produce JavaScript source.

- To build and execute a V program: v run file.v / v –prod run file.v

- To compile all files in the current folder: v .

- To turn on usage of the precompiled module cache: v -cache

This very significantly speeds up secondary compilations:

In Windows: it builds the .o files in C:\Users\CVO\.vmodules\cache\vlib

In Linux: it builds the .o files in $HOME/.vmodules/cache/vlib

(Dec 1: error on Windows

- To obfuscate the resulting binary: v –obf

-To compress the size of the executable: v –compress (?? Oct 31: doesn’t yet work on Windows)

The other options will be discussed in more detail in the linked sections.

**Compiling a module:**

-To build all files in a folder dir into an executable dir (a single main module): v dir

-To compile a module into an object file: v build module mod1 // output is mod1.o

-To build a shared library: v –shared

- ?? To create a library (an object file .o): v -lib mymodule/ or v –lib ./mymodule

here mylib is the module folder and name; command is executed one level above the module’s folder; this creates mymodule.o)

**Installing a module:**

-To install a user module from https://vpm.vlang.io/: v install <module>

- ?? To install a dependency (e.g. sqlite) use v get sqlite

**Creating a project:**

v create project1 Creates a new v project interactively. Answer the questions, and run it with `v run project1`.

It creates a folder project1, in it a V file project1.v and a v.mod file (see ch 11)

**Formatting:** see § 3.3

**Documenting:** see § 3.4

**Testing:**

-To run all V test files located in the folder and its subfolders: v test folder/

You can also pass individual \_test.v files too.

Any file ending in \_test.v, will be treated as a test. It will be compiled and run, evaluating the assert statements in every function named test\_xxx.

-To show additional stats when compiling/running tests: v -stats test .

**Options for debugging/troubleshooting:** see § 3.2

**Environment variable named VFLAGS:**

Common options can be put inside an environment variable named VFLAGS, so that you don't have to repeat them. You can set it like this:

on \*nix (bash):

export VFLAGS="-cc clang -debug"

export VFLAGS="-debug -show\_c\_cmd"

on Windows:

set VFLAGS=-cc msvc

set VFLAGS=-debug -show\_c\_cmd

PowerShell:  $env:VFLAGS="-debug -show\_c\_cmd"

**Cross-compilation:** see § 2.6

**Hot code reloading:** see § 3.11

**Translating C to V:** see § 3.7

## 2.4 Explanations about the compiler

The parser and generator are built on top of an abstract syntax tree (AST). This simplifies the code greatly and allows to implement new backends much faster: the ast compiler generates an ast, then the gen stage uses the ast to generate the C

The source of the new compiler is here: <https://github.com/vlang/v/tree/master/vlib/v>

?? How up to date is the following explanation ??

These are the important files: *parser.v, scanner.v, token.v, gen.v, fn.v,* and *main.v*.

I tried making the code of the compiler and vlib as simple and readable as possible. One of V's goals is to be open to developers with different levels of experience in compiler development. Compilers don't need to be black boxes full of magic that only few people understand.

The compiler itself is located in v/vlib/compiler/=

<https://github.com/vlang/v/tree/master/vlib/compiler>

It has only 8 files (soon to be 7):

1. *main.v* The entry point.

* V figures out the build mode.
* Constructs the compiler object (struct V).
* Creates a list of .v files that need to be parsed.
* Creates a parser object for each file and runs parse() on them (this should work concurrently in the future). The parser emits C or x64 code directly. For performance reasons, there are no intermediate steps (no AST or Assembly code generation).
* If the parsing is successful, a single C file is generated by merging the output from the parsers and carefully arranging all definitions (C is a single pass language).
* Finally, a C compiler is called to compile this C file and generate an executable or a library.

1. *parser.v* The parser. The core of the compiler. This is the largest file (~3.5k loc). parse() method asks the scanner to generate a list of tokens for the file it needs to parse. Then it simply goes through all the tokens one by one.

In V objects can be used before declaration, so there are 2 passes. During the first pass it only looks at declarations and skips function bodies. It memorizes all function signatures, types, consts, etc. During the second pass it looks at function bodies and generates C (e.g. cgen('if ($expr) {') or machine code (e.g. gen.mov(EDI, 1)).

The formatter is embedded in the parser. Correctly formatted tokens are emitted as they are parsed. This allowed to simplify the compiler and avoid duplication, but slowed the compiler down a bit. In the future this will be fixed with build flags and separate binaries for C generation, machine code generation, and formatting. This way there will be no unnecessary branching and function calls.

1. *scanner.v* The scanner's job is to parse a list of characters and convert them to tokens. It also takes care of string interpolation, which is a mess at the moment.
2. *token.v* This is simply a list of all tokens, their string values, and a couple of helper functions.
3. *table.v* V creates one table object that is shared by all parsers. It contains all types, consts, and functions, as well as several helpers to search for objects by name, register new objects, modify types' fields, etc.
4. *cgen.v* The small Cgen struct helps generate C code. It's also shared by all parsers. It has a couple of functions that allow to go back and set something that was previously unknown (like with a := 0 => int a = 0). Some of these functions are hacky and need improvements and simplifications.
5. *fn.v* Handles declaring and calling normal and async functions and methods. This file is about 1000 lines of code, and has some complex logic. It needs to be cleaned up and simplified a bit.
6. *jsgen.v* defines the json code generation. This file will be removed once V supports comptime code generation, and it will be possible to do this using the language's tools.
7. (??) x64/ is the directory with all the machine code generation logic. It will be available in early July. Obviously this is the most complex part of the compiler. It defines a set of functions that translate assembly instructions to machine code, it builds complicated binaries from scratch byte by byte. It manually builds all headers, segments, sections, symtable, relocations, etc. Right now it only has basic support of the x64 platform/Mach-O format (only for macOS), and it can only generate .o files, which then have to be linked with lld.

The rest of the directories are vlib modules: builtin/ (strings, arrays, maps), time/, os/, etc. Their documentation is pretty clear.

An alternative interpreter/compiler is being worked on: <https://github.com/Itay2805/Vork>

## 2.5 Machine code generation

As we saw in § 2.1, V can generate machine code. For now (Nov 22), this is strictly x64 ELF format.

For example: v **-x64** examples/x64/hello\_world.v

If we try this on Windows, we get:

v -x64 can only generate Linux binaries for now. You are not on a Linux system, so you will not be able to run the resulting executable.

x64 elf binary has been successfully generated (in c:\v)

On Linux:

A file out.bin (size 263 bytes) is generated. Chmod +x ./out.bin

Running it with ./out.bin displays the output.

## 2.6 Cross compilation

-To produce an executable for the selected OS: v –os <OS>

<OS> can be linux, mac, windows, msvc.

Use msvc if you want to use the MSVC compiler on Windows.

Examples:

To cross compile your project on a machine running \*nix:  v -os windows .

or  on a machine running Windows:  v -os linux .

No extra steps required, even for GUI and graphical apps!

Nov 26: on Windows:

c:\v\examples>v -os linux hello\_world.v

C:\Users\CVO\AppData\Local\Temp\v\hello\_world.tmp.c: In function 'is\_atty':

C:\Users\CVO\AppData\Local\Temp\v\hello\_world.tmp.c:985:2: warning: control reaches end of non-void function [-Wreturn-type]...

(Use `v -g` to print the entire error message)

V error: C error. This should never happen.

Please create a GitHub issue: https://github.com/vlang/v/issues/new/choose

on Linux: v -os windows hello\_world.v

“/home/ivo/.vmodules/vlib/builtin.o” not found

Build V for Windows on Mac OS: v –o v.exe –embed\_vlib -os windows .

(?? Compiling macOS software only works on macOS for now.)

If you don't have any C dependencies, that's all you need to do. This works even when compiling GUI apps using the ui module or graphical apps using gg.

If you have C dependencies: you will need to install Clang, LLD linker, and download a zip file with libraries and include files for Windows and Linux. V will provide you with a link.

## 2.7 Memory management

There's no garbage collection.

Memory will be freed as soon as it's no longer needed: V is going to use mixed reference counting (kind of GC ??) and compiler handled memory releasing. If compiler can't handle releasing memory by detecting memory is out of scope then reference counting will be used.

V cleans up what it can during compilation. For example:

fn draw\_text(s string, x, y int) {

...

}

fn draw\_scene() {

...

draw\_text('hello $name1', 10, 10)

draw\_text('hello $name2', 100, 10)

draw\_text(strings.repeat('X', 10000), 10, 50)

...

}

The strings don't escape draw\_text, so they are cleaned up when the function exits.

In fact, the first two calls won't result in any allocations at all. These two strings are small, V will use a preallocated buffer for them.

For more complex cases, manual memory management is required. This will be fixed soon.

To clean up, for example, an array numbers, use the free() method:

numbers := [0; 1000000]

...

numbers.free()

Here is another annotated example:

fn test() []int {

number := 7 // stack variable

user := User{} // struct allocated on stack

// array allocated on heap, will be freed as the function exits

numbers := [1, 2, 3]

println(number)

println(user)

println(numbers)

numbers2 := [4, 5, 6] // array that's being returned, won't be freed here

return numbers2

}

V will detect memory leaks at runtime and report them.

What if two objects take a pointer reference to a heap allocated object? who is the owner of that object?

the language will do its best to avoid these if it's unavoidable, then it'll work just like C++ smart pointers

and there will be a way to manage memory directly with unsafe.

- prealloc option for block allocations.

- freestanding: malloc()/free()

?? examples

## 2.8 Bare-metal support

Bare metal support allows programs to build without linking to libc. For this, you have to use the -freestanding flag.

malloc/free functions can be used in this environment.