# Chapter 3 – Editors, IDE’s and tools.

Because V is still very young, there isn’t yet much support in this area, but we’ll expect several possibilities in the not too distant future.

## 3.1 Editors and Integrated Development Environments

?? A Visual IDE (VIDE) for V is planned (work on the IDE will start in August or September).

### 3.1.1. Vid

Vid is a small and fast open source text editor written in V and with the performance of Sublime Text. It is made and used by V’s designer: Alexander Medvednikov. You can find it here: <https://github.com/medvednikov/vid>

Here are its most important features:

* Small size (1 MB text editor written in V with hardware accelerated text rendering. Compiles in <1s).
* Hardware accelerated text rendering
* High performance (scrolling through 300k lines with syntax highlighting without any lag)
* Vim mode
* Easy integration with any compiler/build system
* Go to definition, ctrlp (fuzzy file finder)
* Very fast search in all project files
* Integration with git
* Built-in time management system (based on Pomodoro)
* Global shortcuts (bring to front etc)
* Split view
* Workspaces
* Cross-platform (Windows, macOS, Linux)

Installation:

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

cd vid

v .

./vid

To start with a minimal window and resizing functionality: vid -window .

There are two dependencies: freetype and glfw. (glfw dependency will be removed soon.)

Ubuntu:

sudo apt install libglfw3 libglfw3-dev libfreetype6-dev

macOS:

brew install glfw freetype

Windows:

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

Community: Join the #vid channel on Discord

**How to compile vid with MinGW-w64**. Tested on Windows 7 x64. Assume both the v and vid directories are in the same parent directory

1. Download glfw-3.3.bin.WIN32.zip from <https://www.glfw.org/download.html>

2. Unzip and copy glfw3.dll from folder lib-mingw-w64 to v/thirdparty/glfw (overwrite the existing one) and vid directory

3. mkdir v/thirdparty/freetype

4. Download freetype-windows-binaries-master.zip from <https://github.com/ubawurinna/freetype-windows-binaries>

5. Unzip and copy include and win32 folders to v/thirdparty/freetype

6. Copy freetype.dll from win32 folder to vid directory

7. Add these flags to the end of vid.v:

#flag -I ../v/thirdparty/glad

#flag -I ../v/thirdparty/glfw

#flag -I ../v/thirdparty/stb\_image

#flag -I ../v/thirdparty/freetype/include

#flag -Wl,-L../v/thirdparty/glfw,-lglfw3

#flag -Wl,-L../v/thirdparty/freetype/win32,-lfreetype

8. In the vid directory, v -prod -o vid .

Copy freetype.dll and glfw3.dll to c:\v

* Works on Windows and Linux

Output On Windows:

create window wnd=000000000208BD80 ptr==000000000208AF70

trying to load font from RobotoMono-Regular.ttf

new gg text context vao=3

timer.load\_tasks()

["vid"]

add\_workspace("c:\vid")

load session "C:\Users\CVO\.vid/session"

["", "", "C:\v\vid/out"]

open file "C:\v\vid/out"

getting words

saving session...

KEY UP

How to work with vid:

Instructions: see <https://github.com/medvednikov/vid>

### 3.1.2. Vim

Installation instructions for a Vim plugin are located here: <https://github.com/ollykel/v-vim>  
It's mostly based on go.vim for Golang, with a few modifications. 

REMARK: the file extensions \*.v and \*.vh are also used by Verilog. The installation instructions I provided should override Verilog syntax highlighting, but if it doesn't just enter the command ":setf vlang" after you open your editor.

An alternative Vim syntax highlighter v-vim can be found here: <https://github.com/lcolaholicl/vim-v>

### 3.1.3. Kate

V syntax highlighting for the Kate editor (<https://kate-editor.org/>): <https://github.com/Larpon/kate-syntax-highlight-v>

### 3.1.4 . SciTe

V syntax highlighting for the SciTe editor <https://github.com/sunnylcw/scite-v-support>

### 3.1.5 . Gavecode Studio

This is an upcoming IDE, which will have support for the V language.

<https://www.clenontec.com/blog/gavecode-studio-coming-2020>

<https://www.clenontec.com/en-gb/gavecode/>

### 3.1.6. Visual Studio Code

This is one of the most popular programmer’s editors today (<https://code.visualstudio.com/>)

There is already basic support for V in VS Code through the plugin **vscode-vlang** (v 0.0.8), see

<https://marketplace.visualstudio.com/items?itemName=0x9ef.vscode-vlang>

( source at: <https://github.com/0x9ef/vscode-vlang> )

It provides syntax highlighting and code snippets.

There is also a separate v- code snippets plugin: lorenzopirro.v-snippets (v 0.1.15)

How to edit, build an run a V program in VS-Code through CodeRunner:

Install the *CodeRunner* extension.

File, Preferences, Settings:

Search for: CodeRunner: Custom Command

Set the custom command to run.

Set it to: **v run $fileName**

Set CodeRunner working directory to: /home/user/v\_lang

Close Settings

* Now CTRL+ALT+K or F1 + select “Run Custom Command” compiles and executes the source code in the editor.

### 3.1.7. Visual Studio

?? V 0.1.22: V can now be used with Visual Studio

### 3.1.8. Docker

You can build V with Docker through the following commands:

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

cd v

docker build -t vlang .

docker run --rm -it vlang:latest

v

### 3.1.9. Heroku buildpack for V

See <https://github.com/louis77/heroku-buildpack-v>

## 3.2 Debugging

Application programming needs good debugging support. V offers some useful command-line options in this respect:

- To produce debug info: v –debug file.v

-g Generate debugging information in the backtraces. It has the effect of –debug, and adds V line numbers to the generated executable.

-cg Same as -g, but add C line numbers to the generated executable instead of V line numbers.

Then you can use **gdb** or **lldb** or **ms visual studio** for example or any debugger you would use with C. The generated .c file will also be available (not deleted as normally).

This won’t work nicely because of the V to C translation, but I think there’s a way to fix that. Function names and variables are the same, but line numbers are different.

V will have debugging that logs every time a variable/field was changed.

- To produce verbose output: v –verbose file.v

- To produce the C transpilation, but not compile it: v –o file.c file.v

This is useful, if you want to integrate v as a transpiler into the build system (probably using a Makefile) of an existing large C code base, or if you just want to read the produced C code.

-keep\_c This prevents V from deleting the intermediate .tmp.c file, which is useful if you want to use a debugger like gdb or msvc, when given after -g / -cg

v -show\_c\_cmd file.v Print the full C compilation command and how much time it took.

e.g.: v -show\_c\_cmd hello\_world.v

==========

gcc -std=gnu11 -w -o "hello\_world" "E:\Vlang\The\_Way\_to\_V\Chapter\_3\_IDE\hello\_world.tmp.c"

gcc took 1578 ms

=========

-cc <ccompiler> Specify which C compiler you want to use as a C backend.

The C backend compiler should be able to handle C99 compatible C code.

Common C compilers are gcc, clang, tcc, icc, cl

-cflags <flags> Pass additional C flags to the C backend compiler.

Example: -cflags `sdl2-config --cflags`

?? There’s profiling via the -profile option. Tells you how much time was spent in each function.

Concrete example using lldb:

v -g foo.v && lldb ./foo

with foo = ex1\_debug:

On Windows: (Dec 3)

lldb : command not recognized

**-debug** : C file is not retained in C:\Users\CVO\AppData\Local\Temp\v ??

**-keep\_c**: C file *ex1\_debug.tmp.c* IS retained in C:\Users\CVO\AppData\Local\Temp\v

**-g** : idem, + output:

C compiler=gcc

all .v files:

["c:\v\vlib\builtin\array.v", "c:\v\vlib\builtin\builtin.v", "c:\v\vlib\builtin\builtin\_windows.v", "c:\v\vlib\builtin\cfns.v", "c:\v\vlib\builtin\float.v", "c:\v\vlib\builtin\hashmap.v", "c:\v\vlib\builtin\int.v", "c:\v\vlib\builtin\map.v", "c:\v\vlib\builtin\option.v", "c:\v\vlib\builtin\string.v", "c:\v\vlib\builtin\utf8.v", "c:\v\vlib\strings\builder\_c.v", "c:\v\vlib\strings\similarity.v", "c:\v\vlib\strings\strings.v", "c:\v\vlib\strconv\atoi.v", "ex1\_debug.v"]

**-show\_c\_cmd**

==========

gcc -std=gnu11 -Wall -Wextra -Wno-unused-variable -Wno-unused-parameter -Wno-unused-result -Wno-missing-braces -Wno-unused-label -Werror=implicit-function-declaration -o "ex1\_debug" "C:\Users\CVO\AppData\Local\Temp\v\ex1\_debug.tmp.c" -ldbghelp

gcc took 1407 ms

=========

**-verbose:**

E:\Vlang\The\_Way\_to\_V\Chapter\_3\_IDE\debugging>v -verbose ex1\_debug.v

C compiler=gcc

["v", "-verbose", "ex1\_debug.v"]

all .v files before:

[]

v\_files\_from\_dir ("c:\v\vlib\builtin")

get\_v\_files(ex1\_debug.v)

user\_files:

["ex1\_debug.v"]

>> trying to find strings in E:\Vlang\The\_Way\_to\_V\Chapter\_3\_IDE\debugging\strings ...

>> trying to find strings in E:\Vlang\The\_Way\_to\_V\Chapter\_3\_IDE\debugging\modules\strings ...

>> trying to find strings in c:\v\vlib\strings ...

v\_files\_from\_dir ("c:\v\vlib\strings")

>> trying to find strconv in E:\Vlang\The\_Way\_to\_V\Chapter\_3\_IDE\debugging\strconv ...

>> trying to find strconv in E:\Vlang\The\_Way\_to\_V\Chapter\_3\_IDE\debugging\modules\strconv ...

>> trying to find strconv in c:\v\vlib\strconv ...

v\_files\_from\_dir ("c:\v\vlib\strconv")

imports:

["strings", "strconv"]

all .v files:

["c:\v\vlib\builtin\array.v", "c:\v\vlib\builtin\builtin.v", "c:\v\vlib\builtin\builtin\_windows.v", "c:\v\vlib\builtin\cfns.v", "c:\v\vlib\builtin\float.v", "c:\v\vlib\builtin\hashmap.v", "c:\v\vlib\builtin\int.v", "c:\v\vlib\builtin\map.v", "c:\v\vlib\builtin\option.v", "c:\v\vlib\builtin\string.v", "c:\v\vlib\builtin\utf8.v", "c:\v\vlib\strings\builder\_c.v", "c:\v\vlib\strings\similarity.v", "c:\v\vlib\strings\strings.v", "c:\v\vlib\strconv\atoi.v", "ex1\_debug.v"]

flags=

\* -ldbghelp

cc() isprod=0 outname=ex1\_debug

==========

gcc -std=gnu11 -Wall -Wextra -Wno-unused-variable -Wno-unused-parameter -Wno-unused-result -Wno-missing-braces -Wno-unused-label -Werror=implicit-function-declaration -o "ex1\_debug" "C:\Users\CVO\AppData\Local\Temp\v\ex1\_debug.tmp.c" -ldbghelp

gcc took 1422 ms

On Linux:

sudo apt install lldb

lldb ./ex1\_debug 🡨 starts lldb debugging session

(lldb) run

(lldb) target create ./ex1\_debug

(see debugging/pony-lldb-cheatsheet)

?? none of the commands work

Within V code: println and assert

For now, use of the println statement with string substitution is the simplest way to show the values of your variables.

You can also use the assert function, which is mostly used in the context of testing. With it, you can test simple conditions like here, where we test that variable age has value 63:

Listing 3.1 – assert.v:

fn main() {

age := 63

**assert**(age == 63)

**assert(age == 64)**

}

When the condition in the assert is true, nothing is output; when it is false, a panic is generated and this is shown:

assert.v:4: FAILED: main()

Source:   assert(age == 64)

V panic: An assertion failed.

Linter:

The closest to a linter at this moment (Dec) is:

v -o /tmp/x.c your\_file.v && rm -rf /tmp/x.c

This will show all the v errors; the produced .c code may still contain C errors though. Since it will not actually compile the resulting .c file, it is also faster than a full compilation.

Other features:

* Backtraces are now printed on panics and show source lines (Nov 15: not yet on Windows)
* V also provides interactive debugging support.
* $if debug {} for running code in debug mode only (see § 5.1)
* @FILE, @LINE, @FN, @COLUMN for debugging

Debugging with VS Code on Windows:

Here is some general info about debugging in VSCode: <https://code.visualstudio.com/Docs/editor/debugging>

To debug with gdb install the extension *Native Debug* (<https://marketplace.visualstudio.com/items?itemName=webfreak.debug>). This is a native VSCode debugger, that supports both GDB and LLDB.

To debug in VS Code you need to add a debug configuration launch.json file:

In the menu: Debug / Add Configuration / select GDB

This creates and open *launch.json*. You'll have to manually change the executable name under "target", and check that type is gdb:

{

      "name": "Debug",

      "type": "gdb",

      "request": "launch",

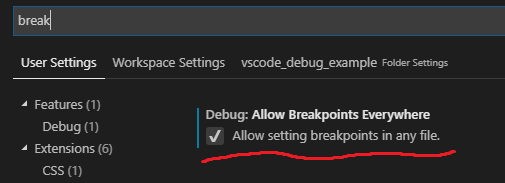
      "target": "${workspaceFolder}/ex1\_debug.exe",

      "cwd": "${workspaceRoot}",

      "valuesFormatting": "parseText"

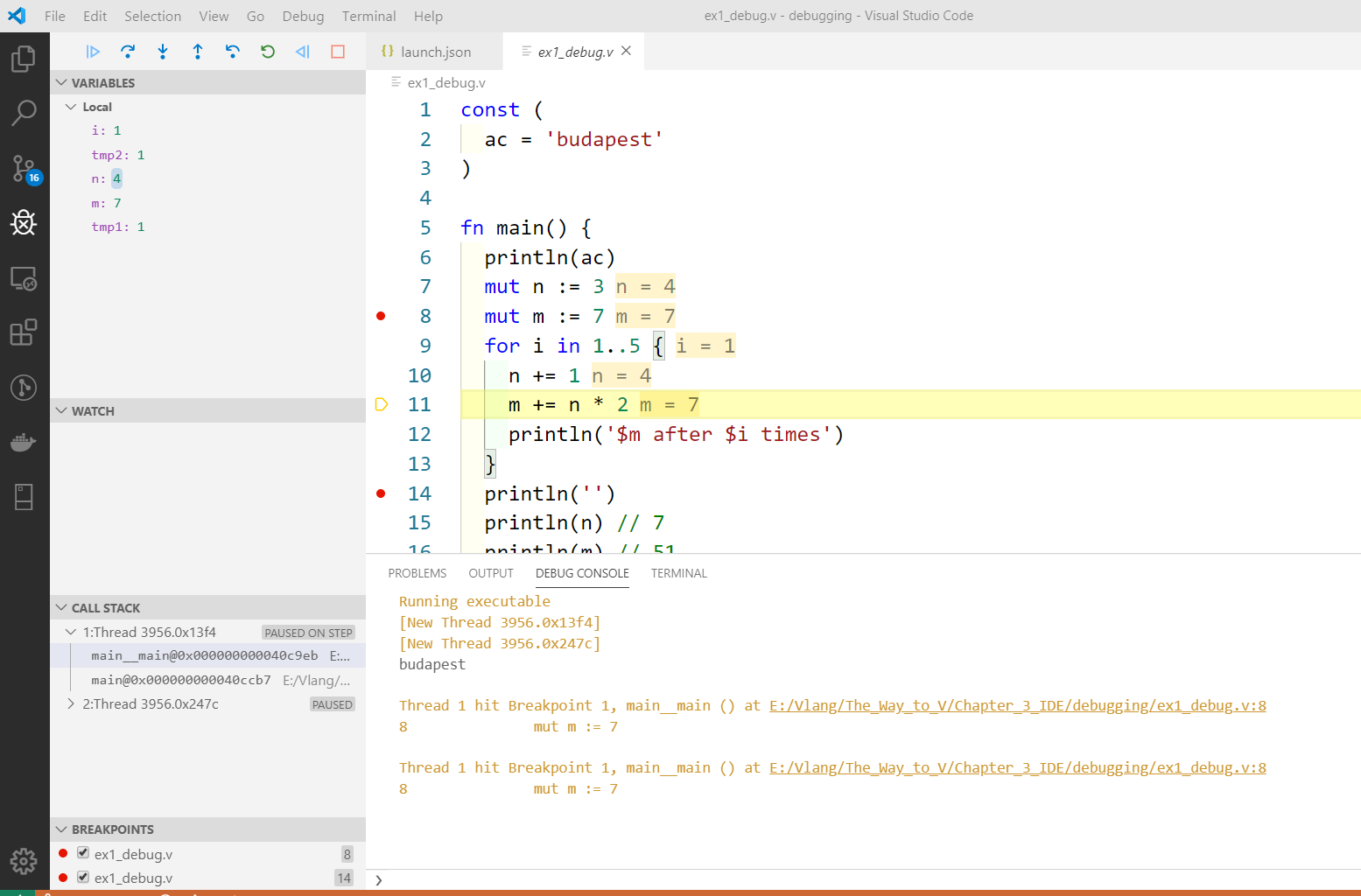
 }

Also, verify that breakpoints are enabled:



Set one or several breakpoints in your code by clicking to the left of the code line(s) where you want to monitor values of variables

Then you can Start Debugging with F5 or from the menu. Below is a screenshot:



## From the debug tool bar, you can step line by line by clicking the arrow icons, or using F11 (Step Into) or F10 (to step over a function).

See the values of the local variables in the left pane, or inside the code window.

F5 continues execution until the next breakpoint, and SHIFT + F5 or the red rectangle stops execution.

## ?? 3.3 Formatting code with vfmt

The V-designer didn’t want endless discussions about code-style for the V-language, discussions which in the past have risen for so many programming languages and that were certainly to an extent a waste of precious development time. So he made a tool: **vfmt**, which provides strict automatic code formatting.

?? (not yet) V has vmft built in: the compiler will automatically update the file.

Vfmt is very strict to ensure one coding style, much stricter than gofmt. It even has a set of rules for empty lines.

*Most editors and IDE’s can be configured so that vfmt runs on every save.*

*!! Always run vfmt before pushing your code. !!*

Using vfmt:

- To run vfmt to format the source code [wip]: v fmt format\_file.v

- ?? To disable automatic formatting: v -nofmt file.v

Test: (Dec 1- Windows):

It says: running vfmt …

But doesn’t appear to do anything: indentation ?, double to single quotes

?? real example

## ?? 3.4 Generating code documentation

The works very similar to Go.

There's no need to write separate documentation for your code, vdoc will generate it from the source code.

Documentation for each function/type/const must be placed right before the declaration:

// clearall clears all bits in the array

fn clearall() {

}

The comment must start with the name of the definition.

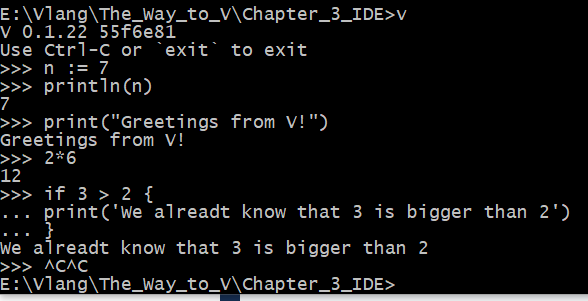
An overview of a module must be placed in the first comment right after the module's name.

?? To generate documentation, run vdoc over the source code v doc path/to/module

## 3.5 REPL

V has a REPL, that starts up when you give the v command without any argument:

v



?? replace with up to date screenshot

An alternative command is: v runrepl or v -

The REPL is an interpreter-like mode takes your code and in the background writes it to a temporary .v file, which is then compiled and run. The output of the program is then displayed in the console. It is compiled fast enough so that the repl is workable, while the state of the session (imports, variables, etc.) remains.

The REPL program lives in c:\v\tools\vrepl.exe; the code of the current session is stored in the file .vrepl\_temp.v

Type help to get an overview of the possible commands.

The UP arrow retrieves the previous command. A clear command clears the screen.

To leave the REPL, give the CTRL/C (Windows) or CTRL/D (\*nix) or exit commands.

The REPL supports imports, consts, function definitions, etc.

Troubleshooting:

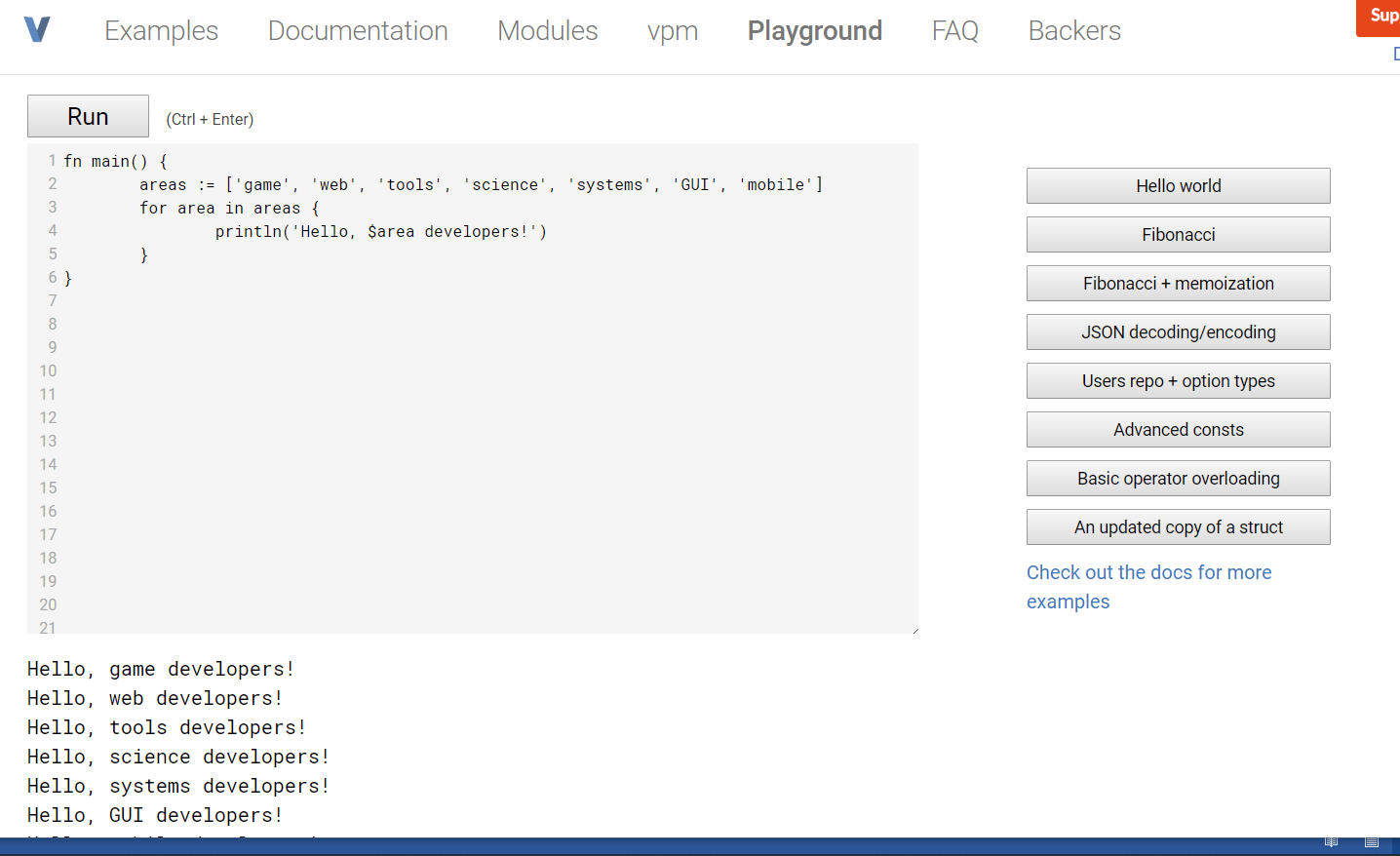
On Windows: If you get this error while running the V REPL, and you are using msvc: 'gcc' is not recognized as an internal or external command, operable program or batch file’, try:

set VFLAGS=-os msvc

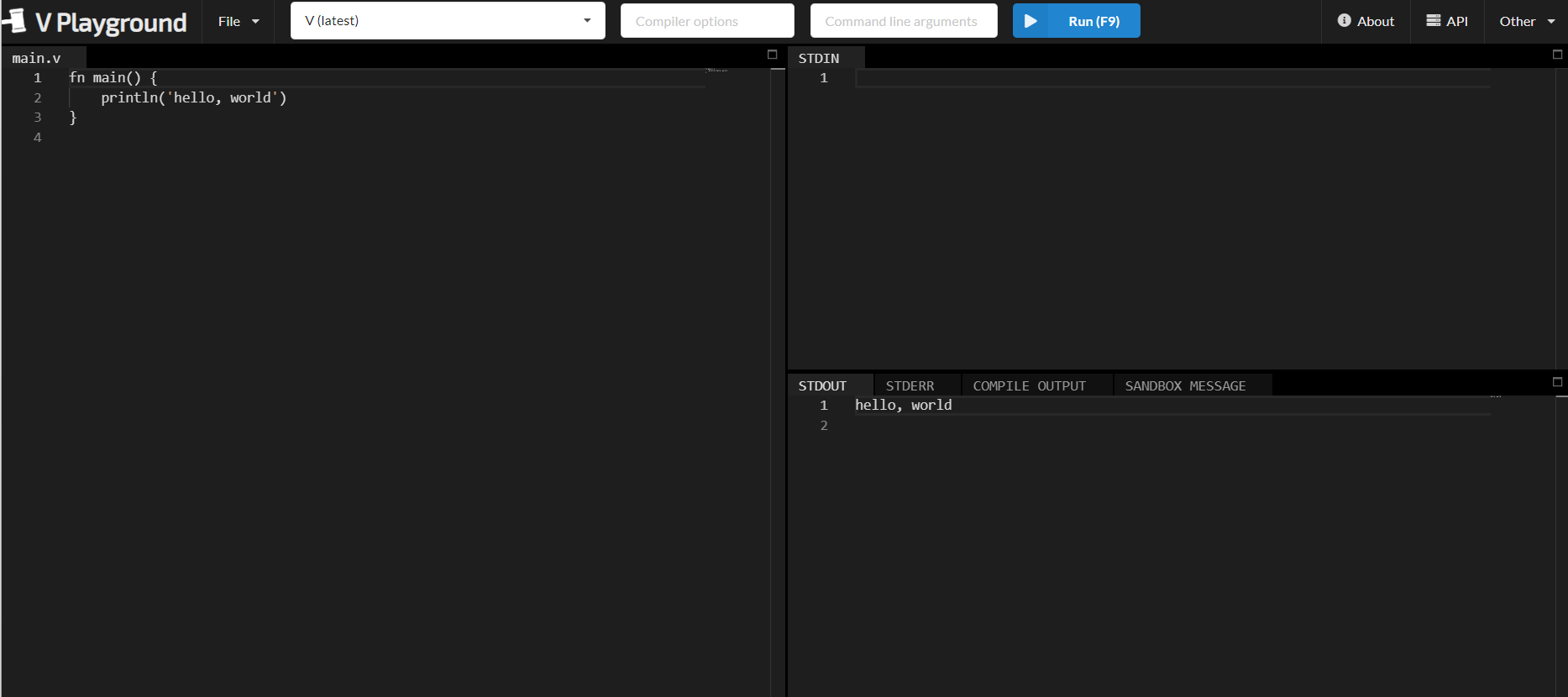
v.exe runrepl

## 3.6 PlayGround

V has an online PlayGround, which is a web page where you can post or edit V code, and then compile and execute it by pushing the Run button or typing F9. It is located here: <https://vlang.io/play>



An alternative with somewhat more facilities is this: <https://vlang.ide.judge0.com/> ; this always runs the latest version of V.



In a future version there will also be a PlayGround that will run in the users browser.

## 3.7 V’s performance

Benchmarks are published here: <https://github.com/kostya/benchmarks>

For example: base64, json, havlak, matmul.

See examples for the use of the benchmark module: base64/test.v

Summary:

## 3.8 Memory management

There's no garbage collection or reference counting. 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.