Tide

Release 0.1

Willem van Ketwich

Jun 29, 2019

CONTENTS:

CHAPTER

ONE

INSTALLATION OF TIDE

In order to install Tide, . . .

CHAPTER

TWO

QUICKSTART GUIDE

The following . . .

CHAPTER

THREE

GDB COMMANDS (REFERENCE GUIDE)

ref: https://sourceware.org/gdb/onlinedocs/gdb/Symbols.html

getting the line and file of the current breakpoint

e.g.

(gdb) info line Line 9 of "hello.c" starts at address 0x40054d <main+39> and ends at 0x40054f.

setting a breakpoint based on file name and line number:

e.g.

gdb> break /Full/path/to/service.cpp:45

getting a list of all variables in current context:

e.g.

info locals

getting a list of all functions:

e.g.

info functions

can take regex:

e.g.

info functions regex

getting all global/local variables:

info variables

getting all variables on the stack frame:

info args

links on variables:

https://stackoverflow.com/questions/6261392/printing-all-global-variables-local-variables

https://sourceware.org/gdb/onlinedocs/gdb/Variables.html

specifying source directories:

https://sourceware.org/gdb/onlinedocs/gdb/Source-Path.html

backtrace also has line numbers:

e.g.

backtrace full

also useful for getting source info:

at current breakpoint:

info source

and

all source files:

info sources

get symbols for a file:

e.g.

maint print symbols -source main.c

a good way to separate external libraries to linked files:

maint info symtabs

the objfile will match the binary being debugged for locally linked files

get a list of lines in each file that breakpoints can be set at:

for all source files:

maint info line-table

or for just one file:

maint info line-table main.c

setting source paths:

https://sourceware.org/gdb/onlinedocs/gdb/Source-Path.html

You can configure a default source path substitution rule by configuring GDB with the '–with-relocated-sources=dir' option. The dir should be the name of a directory under GDB's configured prefix (set with '–prefix' or '–exec-prefix'), and directory names in debug information under dir will be adjusted automatically if the installed GDB is moved to a new location. This is useful if GDB, libraries or executables with debug information and corresponding source code are being moved together.

# Startup process:

after startup:

(gdb) set directories ./tests/binaries/c_test

then

(gdb) info source Current source file is main.c Compilation directory is /binaries/mac_test Located in /work/tests/binaries/c_test/main.c Contains 8 lines. Source language is c. Producer is GNU C11 5.5.0 20171010 -mtune=generic -march=x86-64 -g -fstack-protector-strong. Compiled with DWARF 2 debugging format. Does not include preprocessor macro info.

use 'Located in' line and 'Contains n lines.' lines to verify the source file is correct.

Then display the source file in the primary window.

backtrace - can be used to give the current line number as #0

valid line numbers to set breakpoints can be determined with:

maint info line-table main.c

bash line to run test binary: bin/dev-environment ./tests/binaries/c_test/c_test

# gdb version - needs to be 8.2.1

## Working config at:

plugins/test_c

start:

:Vgdb

set breakpoint to line 6:

:VgRunConfigCommand set_breakpoint

run to breakpoint:

:VgRunConfigCommand run

step through lines:

:VgRunConfigCommand step

CHAPTER

FOUR

SETTING UP KEYS IN VIM

” ide keys: https://delphi.fandom.com/wiki/Default_IDE_Shortcut_Keys ” start nnoremap <F9> :VgRunConfigCommand run<cr> ” continue nnoremap <F8> :VgRunConfigCommand continue<cr> ” step into nnoremap <F7> :VgRunConfigCommand step<cr> ” toggle breakpoint nnoremap <F5> :VgRunConfigCommand set_breakpoint<cr>

# assembly

# test_c

start:

Vgdb

set breakpoint at line 6 of current file:

VgRunConfigCommand set_breakpoint_in_file 6

run:

VgRunConfigCommand run

step:

VgRunConfigCommand step

continue:

VgRunConfigCommand continue

# test_go

CHAPTER

FIVE

VGDB GUIDE

# start Vgdb

:Vgdb

# run a command

:Vgc

# connect to a remote target

:Vgc target remote localhost:9999

# run application and break at entrypoint (sets a breakpoint)

:Vgrte

# display registers window

:Vgreg

# to step into an instruction:

:VgRunConfigCommand stepi

# to continue to end:

:VgRunConfigCommand continue

# to show disassembly:

:Vgdis

# Config description:

commands can have the following types:

type: 'command_with_match' type: 'command' type: 'vim_command' type: 'python_command'

variables can have the following types:

type: 'python_and_vim' type: 'python' type: 'vim'

events can be of the following types:

before_spawn: after_spawn: before_config_command: after_config_command: before_command: after_command: before_buffer_update: after_buffer_update:

event types can be of the following:

type: 'vim' type: 'python'

event types should have the following elements:

run_for: - this is only relevant to commands type: - described above function: the name of the function to run, either python or vim

CHAPTER SIX

STANDARD PROCESS FOR TIDE

The standard process is to have a buffer designated for accepting the output of a command.

The command is run against the process and the output is filtered then placed in the buffer.

A buffer is specified in the config under the buffers section.

A command is specified in the config under the commands section.

A filter is specified in a path (that has been set in the config settings). A filter is for a buffer and will have the same name as the buffer it is providing output for. A base filter may also be provided to do some initial filtering before the buffer's filter does it's work.

For example, the process for running a single command would have the following flow:

Run named config command from Vim -> command is run against running process -> output is filtered with a base filter -> output is filtered with the buffer's filter -> output is displayed in vim

The name of the buffer that the command is for can be set in one of the following ways: - If the command is called from the buffer, e.g. <buffer_name>.command: the buffer name will be passed to the command. - If the command is called as an event after a buffer command is run e.g. <buffer_name>.events: - If the buffer_name is specified explicitly in a run_command config command.

CHAPTER

SEVEN

ACTIONS IN TIDE

There are various built-in actions that can be used to do useful things with vgdb in commands.

The built-in actions consist of:

- create_interpolated_string
- **print_debug** Description: Fields: Example:
- **run_command** Description: Fields: Example:
- **run_command_string** Description: Fields: Example:
- **run_command_with_match** Description: Fields: Example:
- **run_config_command** Description: Fields: Example:
- **run_python_function** Description: Fields: Example:
- **run_vim_funcion** Description: Fields: Example:
- **set_var** Description: Fields: Example:

These are built-in by virtue of the fact that they are in vgdb under the ./actions path. Custom actions can also be created and specified in your config file.

CHAPTER

EIGHT

COMMANDS AND CONFIG COMMANDS, ACTIONS AND EVENTS

variables.user_input_args - can be used in running command string as user input

CHAPTER

NINE

OVERVIEW OF THE KEY CONFIG SECTIONS

buffers:

commands:

events:

settings:

- **buffers:**
  - base_filter_name: is the name of a filter to filter all output with. default is ‘’. if specified, this corresponds with a filter name in the filters path.
  - error_filter_name: is the name of a filter to be applied to output to capture error logs and transform output in some way based on it, if necessary.
  - error_buffer_name: this is the name of an internal buffer to hold the results of the error filter ‘error_filter_name’ in. this is in the config dictionary as vg_config_dictionary[“internal”][“buffers”][error_buffer_name]
  - stack_buffers_by_default: if true, this stacks all new buffers in a single window that splits horizontally with every new buffer.
  - stack_buffer_window_width: this is the width in characters that the buffer will take up of the screen (some of this may need to move once cross-editor is implemented)
  -
- **debugging:**
  - log_to_file: if true, debugging output is logged to a file
  - log_filename: the name of the file for output of vgdb
  - debug_*: all the options starting with **debug_** are various modules to enable debugging for. e.g. debug_command_action is for the module command_action, with the class CommandAction.
- **editor:**
  - name: is the name of the editor that the config will be used for. defaults to ‘vim81’
- **logging:**
  - use_session_log_file: specifies if the output of the session_log will be written to a file. defaults to true.
  - session_log_filename: the name of the file to output the session_log to defaults to ‘vgdb_session.log’
  - session_buffer_name: the name of the buffer to use to store the session_log. defaults to vg_session_log’

  - add_timestamp: if true, adds a timestmp to the session log for every entry. defaults to true.

- **plugins:**
  - actions_path: the location of actions relative to the config path, or absolute. defaults to '../actions'. actions can be used to extend the functionality of commands in the config.
  - filters_path: the location of filters relative to the config path, or absolute. filters are used for capturing specific information from the output for processing in a buffer. defaults to '../filters
  - functions_path: the location of functions that can be called from commands in the config. defaults to '../functions'
- **processes:**
  - ttl_stream_timeout: the time in seconds of how long to wait before timing out on the output of the ttl stream from the process. defaults to 0.08s.
  - run_command_on_startup: if true, runs a command at startup of Vgdb as specified by 'command_to_run_on_startup'
  - command_to_run_on_startup: TODO: requires rework
  - command_on_startup_log_file: TODO: same
  - main_process_name: the name of the process to start for output
  - main_process_default_arguments: a space-separated list of arguments to pass to the main process
  - find_full_process_name: this locates the full path to the process to run if only the name is specified. similar to using 'which'.
  - end_of_output_regex: a regex that determines the end of output after running a command on the process. This is usually for terminal-type applications.

variables:

- contains a list of user-defined variables. these can be accessed in the config and are usually used for specifying defaults for variables and making explicit what variables are being used. This is the main way information is passed between commands. variables can also be updated from filters and functions, or even actions.

CHAPTER

TEN

HOW PLUGIN PATHS WORK

- There is a default config that is called regardless.
- Either the ENV VAR or the config_location.yml file will determine the base config file.
- the vase config file can have a settings.plugins.config_path that specifies a config to be loaded before the base config is loaded. this config can in turn call another config file to be called before it is called.
- The config files are then all merged, starting with the default config file, followed by the last in the config chain all the way up to the config file that was referenced by env var or the config_location.yaml file.
- Then for all the different plugins, the same order for loading is done.
- For things like functions, it pays not to have the same name as a name in a base class as the base version will be called first (may change this).

-All the plugin_paths in the config are first checked as absolute (or relative to the calling path) and then relative to the config that it was declared in. The merge process on config files does not affect the plugin paths in settings.plugins.<plugin>_path.

CHAPTER

ELEVEN

BUFFERS

buffers list the buffers/windows available to be displayed.

a buffer has a set of options:

on_startup: determines whether the buffer starts at startup of :Vgdb command: a config command that can be run for the buffer every time an action occurs line_numbers: determines whether the buffer will have line numbers or not buffer_filename_variable: if set, specifies the name of a variable to use for specifying the filename of the buffer. if not specified, the buffer will be named as per the config file designation. primary_window: if true, the buffer will be the primary window in the editor and will not be stacked with other windows. There should be only one primary window. The primary window is usually used for code editing. Non-primary windows are for more trivial (but still important) windows. language: defines the syntax language to use for syntax highlighting in the buffer.

events: these can happen either: before_command after_command

and are extra commands that run before/after the *command:* command has run. Each command can be presupposed with 'input_args:' which are a list of key-values that are sent to the command for further processing depending on the context. Usually used when running python or vim functions.

CHAPTER TWELVE

HOW FILTERS WORK:

base filter

error filter

buffer filters

session_log filter

how to implement a filter:

filter sections:

CHAPTER THIRTEEN

TYPES

The config file has a set of config commands that make up a config_command_item.

these config commands have a series of steps, or config_command_actions, or just command_actions.

Each command_action has a type (as per the /actions path.

A command_action is a single step in a config_command_item.

A command_action that runs against the process is just a command.

relationship is:

config - command - action

—

buffers may have one or more filter

filters can apply to all output or just one buffer

events:

event_input_args - used from events in config (input_args) - eg. do_buffer_diff - that is run from a buffer event command_args - used from commands (internal) for additional data for the command that is run from an event(lines, command string, etc.) - eg. set_remote_target could standardise command_args to to use a buffer

event flows:

buffer: user action - > config -> command -> buffer -> buffer_event (event_input_args)

user_action: user action -> config -> command -> config event -> command event (command_args)

event types:

buffer updates: - before_buffer - after_buffer

user actions: - before_startup - afer_startup - before_command - after_command