Contents

LLDB Debugger in MacOS for CPP - Using lldb command line

keywords

#debug, #cpp , #lldb, #cli, #debugging,

Content

How to run a debugging in command line using the `lldb` which is used to understand more about how the program is running. I will demonstrate here with an example to make it easy to be used later.

Example Project

- ☒ Assume we are having the following structure. [[Pasted image 20230824233713.png]]

Step-1 Build your binary

- ☒ It is very important to use the flag `-g` for your complier, later I discovered that if you don't use it the setting breakpoint for the `lldb` will not work. Its crucial to add this flag > When you don't compile with the `-g` flag, the generated binary lacks all this essential debug information. Consequently, trying to set a breakpoint on a specific line of source code would be meaningless to the debugger, as it wouldn't have the mapping from that line to a specific location in the binary. That's why you encounter issues when trying to debug a binary that hasn't been compiled with debug information.

```
clang++ -std=c++17 -g ./src/main.cpp -o ./build/debug/main && ./build/debug/main
```

- Notice how we added the flag `-g` as we stated.
  - This flag is automatic in `Rust` which is not available in `CPP`, so we have to add it manually.

Step-2 Running the LLDB

- ☒ Here you have two options first either you provide the `binary` at the starting of the `lldb` as shown below

```
lldb ./build/debug/main
```

- ☒ Or by running the CLI inside

```
(lldb) target create ./build/debug/main
```

- ☒ It is much effective at this point to run while inside the lldb the value
  - ☒ Use `r` or `run` after you created the target, this will ensure to run the binary and connect it to your files.
  - ☒ Always you can use `tab` to autocomplete the command and seek other options offered by the `lldb`.

Step-3 Set a breakpoint

Setting a breakpoint will work once you built your binary using the `-g` flag, and it will offer you also auto-complete to your commands and will understand where to find the files.

- ☒ Setting a breakpoint can be done by several ways, the easiest way is to use:
  - ☒ I am running the following command at the root directory, so it will understand there is a file inside `src` directory even if I don't specify as you can see in the command below:

```
(lldbinit) breakpoint set --file main.cpp --line 83
```

- ☒ You have to be sure about where to set your breakpoint, usually inside a `for-loop` so we can track the changes, and not exit immediately once we run.
  - ☒ There is also another way if you have several files and you want to set a breakpoint at specific function that you know the function name, you can use

```
(lldbinit) breakpoint set --name <function_name>
# Example
(lldbinit) breakpoint set --name main
```

- ☒ To list all breakpoints you can use

```
(lldb) breakpoint list
```

Setup-4 Run again

- ☒ Run the `lldb` again and it will run and stop at the breakpoint that you set it ,
  - ☒ There are two commands that I use often, these are `print <variable>` or `p var` which will show the value, can also be used for expression,
  - ☒ Also I read the memory address of a given variable using `memory read &variable` (usually it is necessary to use the `&` in front the

variable name) ## Other command line for lldb very much useful The following commands are very much useful and I use them a lot when I run debugging in `C++`,

Command Group	Command	Description
Program Execution	`run` or `r`	Start or restart the debugged program.
	`continue` or `c`	Continue execution after stopping at a breakpoint.
	`step` or `s`	Execute next line, step into functions.
	`next` or `n`	Execute next line, step over functions.
	`finish` or `f`	Execute until the current function completes.
	`kill`	Terminate the running program.
Breakpoints	`breakpoint set --name <function_name>`	Set breakpoint on a function.
	`breakpoint set --file <file_name> --line <line_number>`	Set breakpoint on a specific line.
	`breakpoint list`	List all breakpoints.
	`breakpoint delete <breakpoint_id>`	Remove a breakpoint.
	`breakpoint enable <breakpoint_id>`	Enable a breakpoint.
	`breakpoint disable <breakpoint_id>`	Disable a breakpoint without deleting it.
Inspecting State	`print <expression>` or `p <expression>`	Print value of an expression/variable.
	`frame variable` or `fr v`	Display local variables.
	`frame select <index>`	Select a different frame.
Stack & Thread	`thread list`	List all threads.
	`thread select <thread_id>`	Switch to a different thread.
	`bt` or `backtrace`	Display the current thread's call stack.
Memory & Registers	`memory read <address>` or `x <address>`	Read memory from an address.
	`register read`	Display all registers.

Command Group	Command	Description
	`register read <register_name>`	Display a specific register.
File & Source Navigation	`list`	Display source code around current line.
	`list <file_name>:<line_number>`	Display source code around specified line.
Watchpoints	`watchpoint set variable <variable_name>`	Set watchpoint on a variable.
	`watchpoint list`	List all watchpoints.
	`watchpoint delete <watchpoint_id>`	Remove a watchpoint.
Handling Input/Output	`process launch -- <args>`	Start program with arguments.
	`process handle <signal_name> --stop`	Stop program on a specific signal.
Miscellaneous	`help`	Display command help.
	`quit` or `q`	Exit LLDB.
	`settings set target.input-path <file>`	Redirect input from a file.
	`settings set target.output-path <file>`	Redirect output to a file.
Loading Scripts & Extensions	`command script import <path_to_python_script>`	Load a Python script to extend LLDB functionalities.

This table provides an organized reference for the common LLDB commands. Remember, you can always use the `help` command in LLDB for more detailed information on any specific command.