with-qwen

help me distinguish between pure lua programming apis and neovim lua apis

help me learn lua language by code examples

help me know neovim lua apis by code examples

imagine we make neovim plugin with c-program as backend, explain to me

---

I'll help you distinguish between pure Lua programming APIs and Neovim-specific Lua APIs. Understanding this difference is crucial for writing correct Neovim configurations and plugins.

Pure Lua APIs

These are standard Lua language features and libraries that work in any Lua environment:

Core Language Features

```lua
-- Variables, functions, control structures
local x = 10
function my_func() end
if condition then end
for i = 1, 10 do end
while condition do end
```

Standard Libraries

```lua
-- Basic library (type, tostring, etc.)
print("Hello")
type(my_var)
tostring(my_var)
tonumber("123")

-- Table library
table.insert(tbl, item)
table.remove(tbl, index)
table.concat(tbl, sep)

-- String library
string.match(str, pattern)
string.gsub(str, pattern, replacement)
string.format("Hello %s", name)

-- Math library
math.random()
math.floor(x)
math.max(a, b)

-- OS library
```

```
os.date()
os.getenv("HOME")
os.execute("ls")

-- IO library
io.open("file.txt", "r")
file:read("*a")
```

Neovim Lua APIs

These are Neovim-specific APIs that only exist within Neovim:

Core Neovim APIs

```
-- Neovim API functions (vim.api)
vim.api.nvim_buf_get_lines(0, 0, -1, false)
vim.api.nvim_win_get_cursor(0)
vim.api.nvim_exec("normal! gg", false)

-- Shortcuts for common operations
vim.cmd("set number")
vim.fn.getcwd()
vim.g.global_variable = "value"  -- Global variables
vim.o.number = true             -- Options
vim.bo.modified = false         -- Buffer-local options
vim.wo.number = true            -- Window-local options
```

Keymap and Command APIs

```
-- Key mappings
vim.keymap.set('n', '<leader>s', ':w<CR>')
vim.api.nvim_set_keymap('n', '<leader>q', ':q<CR>', {noremap = true})

-- Commands
vim.api.nvim_create_user_command('MyCommand', function(opts)
    print("Command executed")
end, {})
```

Plugin and Runtime APIs

```
-- Plugin management
vim.opt.runtimepath:append("/path/to/plugin")

-- Package management
vim.cmd("packadd! plugin-name")

-- Autocmds
vim.api.nvim_create_autocmd("BufEnter", {
    pattern = "*.lua",
    callback = function() print("In Lua file") end
})
```

Quick Identification Tips

Pure Lua APIs:

- Start with standard Lua library names: `string.` , `table.` , `math.` , `os.` , `io.`
- Basic language constructs: `function` , `if` , `for` , `while` , `local`
- `require()` for module loading (though Neovim extends this)

Neovim APIs:

- Always involve `vim.` prefix
- `vim.api.nvim_*` functions
- `vim.fn.*` (for VimL functions)
- `vim.cmd()` for executing Vim commands
- `vim.keymap` , `vim.diagnostic` , `vim.lsp` , etc.
- Any API that interacts with Neovim's editor functionality

Hybrid Example:

```
-- Pure Lua: creating a table
local my_commands = {}

-- Neovim API: creating a command
vim.api.nvim_create_user_command('MyCmd', function()
    -- Pure Lua: string concatenation
    local msg = "Hello " .. "World"
    -- Neovim API: displaying message
    vim.notify(msg)
end, {})
```

Testing Environment:

If you try to run `vim.api.nvim_buf_get_lines()` in a standalone Lua interpreter, it will fail because `vim` doesn't exist outside of Neovim. Conversely, Neovim fully supports all standard Lua features.

The key distinction is: **Pure Lua works anywhere Lua runs**, while **Neovim APIs only work inside Neovim**.

Learning Lua Through Code Examples

Let me guide you through Lua programming with practical examples, from basic to advanced concepts.

1. Basic Syntax and Variables

```
-- Single line comment
--[[
```

```lua
   Multi-line comment
   This is very useful for documentation
--]]

-- Variables (no declaration needed)
local name = "Alice"        -- local variable (recommended)
global_var = 42            -- global variable (use sparingly)
local pi = 3.14159
local is_valid = true
local nothing = nil        -- represents "no value"

print(name, pi, is_valid)  -- Output: Alice 3.14159 true
```

2. Data Types

```lua
-- Numbers (all are double-precision floating-point)
local integer = 42
local float = 3.14
local scientific = 1.2e-4

-- Strings
local single_quotes = 'Hello'
local double_quotes = "World"
local multi_line = [[
This is a
multi-line string
]]

-- String operations
local full = single_quotes .. " " .. double_quotes  -- concatenation
local length = #full                                -- length
local upper = string.upper("hello")                -- "HELLO"

-- Booleans and nil
local truthy = true
local falsy = false
local nothing = nil

print("Type of 42:", type(42))           -- "number"
print("Type of hello:", type("hello"))   -- "string"
print("Type of nil:", type(nil))         -- "nil"
```

3. Tables (Lua's Only Data Structure)

```lua
-- Array-like table (1-indexed!)
local fruits = {"apple", "banana", "orange"}
print(fruits[1])  -- "apple" (first element)
```

```lua
-- Associative array (dictionary/hash)
local person = {
    name = "John",
    age = 30,
    city = "New York"
}
print(person.name)     -- "John"
print(person["age"])   -- 30

-- Mixed table
local mixed = {
    [1] = "first",
    ["key"] = "value",
    method = function() return "hello" end
}
print(mixed[1])        -- "first"
print(mixed.method())  -- "hello"

-- Table operations
table.insert(fruits, "grape")          -- add to end
table.insert(fruits, 1, "mango")       -- insert at position 1
table.remove(fruits, 2)                -- remove element at index 2
print(#fruits)                         -- get length
```

4. Control Structures

```lua
-- If statements
local score = 85
if score >= 90 then
    print("A")
elseif score >= 80 then
    print("B")
else
    print("C")
end

-- While loop
local i = 1
while i <= 3 do
    print("While loop:", i)
    i = i + 1
end

-- Numeric for loop (1-indexed!)
for i = 1, 5 do
    print("For loop:", i)
end

-- Generic for with pairs (key-value iteration)
local colors = {red = "#FF0000", green = "#00FF00", blue = "#0000FF"}
```

```lua
for key, value in pairs(colors) do
    print(key, value)
end

-- Generic for with ipairs (array iteration)
local items = {"apple", "banana", "orange"}
for index, value in ipairs(items) do
    print(index, value)
end

-- Repeat-until loop
local count = 1
repeat
    print("Repeat:", count)
    count = count + 1
until count > 3
```

5. Functions

```lua
-- Basic function
function greet(name)
    return "Hello, " .. name .. "!"
end

print(greet("Alice"))  -- "Hello, Alice!"

-- Local function
local function calculate_area(length, width)
    return length * width
end

-- Function with multiple returns
function divide(a, b)
    if b == 0 then
        return nil, "Division by zero"
    end
    return a / b, a % b  -- quotient, remainder
end

local result, remainder = divide(10, 3)
print(result, remainder)  -- 3.3333333333333 1

-- Anonymous function (function literal)
local square = function(x) return x * x end
print(square(5))  -- 25

-- Variadic function (variable arguments)
function sum(...)
    local args = {...}
    local total = 0
    for _, value in ipairs(args) do
        total = total + value
    end
    return total
end

print(sum(1, 2, 3, 4, 5))  -- 15

-- Function with default parameters
function create_user(name, age, city)
    age = age or 18        -- default to 18 if nil
    city = city or "Unknown"
    return {name = name, age = age, city = city}
end

local user = create_user("Bob")  -- age defaults to 18, city to "Unknown"
```

6. Advanced Table Usage

```
-- Nested tables (like objects/structs)
local game = {
    player = {
        name = "Hero",
        level = 1,
        stats = {
            health = 100,
            mana = 50
        }
    },
    inventory = {
        weapons = {"sword", "shield"},
        items = {"potion", "key"}
    }
}

print(game.player.name)                    -- "Hero"
print(game.player.stats.health)           -- 100
print(game.inventory.weapons[1])          -- "sword"

-- Metatables (for custom behavior)
local Vector = {}
Vector.__index = Vector

function Vector:new(x, y)
    local obj = {x = x, y = y}
    setmetatable(obj, Vector)
    return obj
end

function Vector:add(other)
```

```
    return Vector:new(self.x + other.x, self.y + other.y)
end

local v1 = Vector:new(1, 2)
local v2 = Vector:new(3, 4)
local v3 = v1:add(v2)
print(v3.x, v3.y)  -- 4 6
```

7. Error Handling

```
-- Using pcall (protected call)
local success, result = pcall(function()
    return 10 / 2
end)

if success then
    print("Result:", result)  -- "Result: 5"
else
    print("Error:", result)
end

-- Error handling with custom error
function safe_divide(a, b)
    if b == 0 then
        error("Cannot divide by zero!")
    end
    return a / b
end

local success, result = pcall(safe_divide, 10, 0)
if not success then
    print("Caught error:", result)  -- "Caught error: Cannot divide by zero!"
end
```

8. Modules and require()

```
-- math_utils.lua (create this file)
local M = {}

function M.square(x)
    return x * x
end

function M.cube(x)
    return x * x * x
end

M.PI = 3.14159
```

```lua
return M

-- In main script
local math_utils = require("math_utils")  -- assumes math_utils.lua exists
print(math_utils.square(4))  -- 16
print(math_utils.PI)         -- 3.14159
```

9. Practical Example: Simple Todo List

```lua
-- Simple todo list application
local TodoList = {}
TodoList.__index = TodoList

function TodoList:new()
    local obj = {
        items = {},
        next_id = 1
    }
    setmetatable(obj, TodoList)
    return obj
end

function TodoList:add_task(description)
    local task = {
        id = self.next_id,
        description = description,
        completed = false,
        created_at = os.date("%Y-%m-%d %H:%M:%S")
    }
    table.insert(self.items, task)
    self.next_id = self.next_id + 1
    return task.id
end

function TodoList:complete_task(task_id)
    for _, task in ipairs(self.items) do
        if task.id == task_id then
            task.completed = true
            return true
        end
    end
    return false
end

function TodoList:list_tasks()
    for _, task in ipairs(self.items) do
        local status = task.completed and "[x]" or "[ ]"
        print(status, task.id, "-", task.description, "(" .. task.created_at .. ")")
    end
```

```
end

-- Usage
local todos = TodoList:new()
todos:add_task("Learn Lua")
todos:add_task("Write code")
todos:add_task("Practice examples")
todos:complete_task(1)
todos:list_tasks()
```

10. Useful Lua Patterns

```
-- Safe table access (nil-safe navigation)
local function safe_get(t, ...)
    local keys = {...}
    for i = 1, #keys do
        if t == nil then return nil end
        t = t[keys[i]]
    end
    return t
end

local config = {database = {host = "localhost"}}
print(safe_get(config, "database", "host"))     -- "localhost"
print(safe_get(config, "database", "port"))     -- nil (no error)

-- Table deep copy
local function deepcopy(orig)
    local orig_type = type(orig)
    local copy
    if orig_type == 'table' then
        copy = {}
        for orig_key, orig_value in next, orig, nil do
            copy[deepcopy(orig_key)] = deepcopy(orig_value)
        end
        setmetatable(copy, deepcopy(getmetatable(orig)))
    else
        copy = orig
    end
    return copy
end

-- IIFE (Immediately Invoked Function Expression)
local result = (function()
    local temp = 10
    return temp * 2
end)()
print(result)  -- 20
```

Key Lua Concepts to Remember:

1. **Everything is a table** - even functions and modules
2. **1-indexed arrays** - unlike most programming languages
3. **Dynamic typing** - no type declarations needed
4. **First-class functions** - functions can be stored in variables
5. **Lexical scoping** - use `local` for block scope
6. **Garbage collection** - automatic memory management

Practice these examples and gradually build more complex programs to master Lua!

Neovim Lua APIs - Comprehensive Guide with Code Examples

Let me show you the essential Neovim Lua APIs with practical examples, organized by functionality.

1. Basic Neovim API Access

```lua
-- All Neovim APIs are accessed through vim.* namespace
print(vim.api.nvim_get_current_buf())  -- Get current buffer handle
print(vim.api.nvim_get_current_win())  -- Get current window handle
print(vim.api.nvim_buf_line_count(0))  -- Get line count of current buffer (0 = current)

-- Shortcuts for common operations
vim.cmd("set number")           -- Execute Vim command
local cwd = vim.fn.getcwd()     -- Call VimL function
vim.g.my_global = "value"       -- Set global variable
vim.o.number = true             -- Set option
vim.bo.filetype = "lua"         -- Set buffer-local option
vim.wo.number = true            -- Set window-local option
```

2. Buffer Operations

```lua
-- Create and manipulate buffers
local buf = vim.api.nvim_create_buf(false, true)  -- scratch buffer (not listed, scratch)
vim.api.nvim_buf_set_name(buf, "MyBuffer")
vim.api.nvim_buf_set_lines(buf, 0, -1, false, {"Line 1", "Line 2", "Line 3"})

-- Get buffer information
local current_buf = 0  -- 0 means current buffer
local lines = vim.api.nvim_buf_get_lines(current_buf, 0, -1, false)  -- Get all lines
local buf_name = vim.api.nvim_buf_get_name(current_buf)
local buf_changed = vim.api.nvim_buf_get_option(current_buf, "modified")
```

```lua
-- Set buffer options
vim.api.nvim_buf_set_option(current_buf, "buftype", "nofile")
vim.api.nvim_buf_set_option(current_buf, "filetype", "markdown")

-- Buffer keymaps
vim.api.nvim_buf_set_keymap(current_buf, 'n', '<leader>q', ':q<CR>', {
    noremap = true,
    silent = true
})
```

3. Window Operations

```lua
-- Get window information
local current_win = vim.api.nvim_get_current_win()
local win_width = vim.api.nvim_win_get_width(current_win)
local win_height = vim.api.nvim_win_get_height(current_win)
local win_buf = vim.api.nvim_win_get_buf(current_win)

-- Set window options
vim.api.nvim_win_set_option(current_win, "number", true)
vim.api.nvim_win_set_option(current_win, "relativenumber", true)

-- Get/set window cursor position
local cursor_pos = vim.api.nvim_win_get_cursor(current_win)  -- {row, col} (1-indexed)
vim.api.nvim_win_set_cursor(current_win, {10, 5})  -- Move to line 10, column 5

-- Create new window
local new_buf = vim.api.nvim_create_buf(false, true)
local new_win = vim.api.nvim_open_win(new_buf, false, {
    relative = 'editor',
    width = 60,
    height = 10,
    row = 5,
    col = 5,
    style = 'minimal'
})
```

4. Keymap Management

```lua
-- Modern keymap API (recommended)
vim.keymap.set('n', '<leader>w', ':w<CR>', { desc = "Save file" })
vim.keymap.set('i', 'jk', '<Esc>', { desc = "Exit insert mode" })
vim.keymap.set('n', '<leader>q', ':q<CR>', {
    desc = "Quit buffer",
    silent = true,
    noremap = true
})
```

```
-- Buffer-local keymap
vim.keymap.set('n', '<leader>c', function()
    print("Current line: " .. vim.api.nvim_get_current_line())
end, { buffer = 0, desc = "Show current line" })

-- Multiple modes
vim.keymap.set({'n', 'v'}, '<leader>y', '"+y', { desc = "Copy to system clipboard"
})

-- Remove keymap
vim.keymap.del('n', '<leader>w')
```

5. Command Creation

```
-- Create user commands
vim.api.nvim_create_user_command('Hello', function(opts)
    vim.notify("Hello, " .. (opts.args or "World") .. "!")
end, {
    nargs = '?',  -- 0 or 1 argument
    desc = 'Say hello with optional name'
})

vim.api.nvim_create_user_command('Grep', function(opts)
    -- Execute external command and capture output
    local handle = io.popen('grep -n "' .. opts.args .. '" ' .. vim.fn.expand('%'))
    local result = handle:read('*a')
    handle:close()
    print(result)
end, { nargs = 1 })

-- Command with completion
vim.api.nvim_create_user_command('MyFile', function(opts)
    vim.cmd('edit ' .. opts.args)
end, {
    nargs = 1,
    complete = 'file'  -- File completion
})
```

6. Autocmds (Event Handling)

```
-- Create autocommands
local augroup = vim.api.nvim_create_augroup('MyAutoGroup', { clear = true })

vim.api.nvim_create_autocmd('BufWritePre', {
    group = augroup,
    pattern = '*.lua',
    callback = function()
```

```lua
        print("About to save a Lua file!")
    end
})

-- Multiple events
vim.api.nvim_create_autocmd({'BufEnter', 'WinEnter'}, {
    group = augroup,
    pattern = '*.py',
    callback = function()
        vim.opt_local.expandtab = true
        vim.opt_local.shiftwidth = 4
    end
})

-- FileType specific
vim.api.nvim_create_autocmd('FileType', {
    group = augroup,
    pattern = 'markdown',
    callback = function()
        vim.opt_local.wrap = true
        vim.opt_local.linebreak = true
    end
})

-- Clean up autocommands when plugin is unloaded
vim.api.nvim_create_autocmd('User', {
    group = augroup,
    pattern = 'VeryLazy',
    callback = function()
        -- Your initialization code here
        vim.cmd('highlight Comment cterm=italic gui=italic')
    end
})
```

7. Option Management

```lua
-- Global options
vim.opt.number = true
vim.opt.relativenumber = true
vim.opt.expandtab = true
vim.opt.tabstop = 4
vim.opt.shiftwidth = 4

-- List options (append/remove)
vim.opt.runtimepath:append("/path/to/plugin")
vim.opt.path:remove("/unwanted/path")

-- Buffer-local options
vim.opt_local.buftype = "nofile"
vim.opt_local.bufhidden = "hide"
```

```lua
-- Window-local options
vim.wo.number = true
vim.wo.signcolumn = "yes"

-- Get option values
local num = vim.opt.number:get()
local tabstop = vim.opt.tabstop:get()
```

8. Diagnostic API (LSP Diagnostics)

```lua
-- Show diagnostics in a floating window
local function show_diagnostics()
    local diagnostics = vim.diagnostic.get(0)  -- Get diagnostics for current buffer
    for _, d in ipairs(diagnostics) do
        print(string.format("%s:%d:%d: %s (%s)",
            vim.api.nvim_buf_get_name(d.bufnr),
            d.lnum + 1,  -- Convert to 1-indexed
            d.col + 1,
            d.message,
            vim.diagnostic.severity[d.severity]
        ))
    end
end

-- Navigate diagnostics
vim.keymap.set('n', ']d', function() vim.diagnostic.goto_next() end, { desc = "Next
diagnostic" })
vim.keymap.set('n', '[d', function() vim.diagnostic.goto_prev() end, { desc =
"Previous diagnostic" })

-- Show diagnostic in floating window
vim.keymap.set('n', 'gl', function() vim.diagnostic.open_float() end, { desc = "Show
diagnostic" })

-- Configure diagnostic display
vim.diagnostic.config({
    virtual_text = true,
    signs = true,
    update_in_insert = false,
    underline = true,
})
```

9. LSP Client API

```lua
-- Configure LSP servers
local lsp = vim.lsp
```

```
-- Attach a handler to LSP events
lsp.handlers['textDocument/hover'] = lsp.with(lsp.handlers.hover, {
    border = 'rounded',
    max_width = 80,
})

-- Get LSP client information
local clients = vim.lsp.get_clients()
for _, client in ipairs(clients) do
    print("Client:", client.name, "ID:", client.id)
end

-- Send request to specific client
local client = vim.lsp.get_client_by_id(1)
if client then
    client.request('textDocument/completion', {
        textDocument = vim.lsp.util.make_text_document_params(),
        position = vim.lsp.util.make_position_params().position,
    }, function(err, result)
        if not err then
            print(vim.inspect(result))
        end
    end)
end
```

10. Treesitter API

```
-- Get treesitter parser for current buffer
local parser = vim.treesitter.get_parser(0, vim.bo.filetype)
local tree = parser:parse()[1]
local root = tree:root()

-- Query treesitter nodes
local query = vim.treesitter.query.parse(vim.bo.filetype, [[
    (function_definition
        name: (identifier) @function-name)
]])
for id, node in query:iter_captures(root, 0, 0, -1) do
    local name = vim.treesitter.get_node_text(node, 0)
    print("Function:", name)
end
```

11. UI Elements (Floating Windows, etc.)

```
-- Create a floating window
local function create_floating_window()
    local buf = vim.api.nvim_create_buf(false, true)
    local width = 60
    local height = 10

    local row = (vim.o.lines - height) / 2
    local col = (vim.o.columns - width) / 2

    local win = vim.api.nvim_open_win(buf, true, {
        relative = 'editor',
        width = width,
        height = height,
        row = math.floor(row),
        col = math.floor(col),
        style = 'minimal',
        border = 'rounded'
    })

    -- Set content
    vim.api.nvim_buf_set_lines(buf, 0, -1, false, {
        "Floating Window",
        "Press q to close",
        "",
        "This is a floating window!"
    })

    -- Set keymap to close
    vim.api.nvim_buf_set_keymap(buf, 'n', 'q', '<cmd>close<CR>', {
        noremap = true,
        silent = true
    })

    return win, buf
end

-- Create floating window on command
vim.api.nvim_create_user_command('FloatWin', create_floating_window, {})
```

12. Plugin Development Example

```
-- Complete plugin example: Simple status line
local M = {}

function M.setup()
    -- Set up autocommands
    local augroup = vim.api.nvim_create_augroup('SimpleStatusLine', { clear = true
})

    vim.api.nvim_create_autocmd({'BufEnter', 'WinEnter', 'FileType'}, {
        group = augroup,
        callback = function()
            vim.opt_local.statusline = '%f %m %= Line %l/%L Col %c'
```

```lua
        end
    })

    -- Add some utility functions
    vim.api.nvim_create_user_command('ToggleNumbers', function()
        vim.wo.number = not vim.wo.number
        vim.wo.relativenumber = vim.wo.number
    end, {})
end

-- Usage: require('myplugin').setup()
return M
```

13. Notification System

```lua
-- Different types of notifications
vim.notify("Simple message", vim.log.levels.INFO)
vim.notify("Warning message", vim.log.levels.WARN)
vim.notify("Error message", vim.log.levels.ERROR)
vim.notify("Debug message", vim.log.levels.DEBUG)

-- Custom notification with options
vim.notify("Custom notification", vim.log.levels.INFO, {
    title = "My Plugin",
    on_open = function(win)
        vim.api.nvim_win_set_option(win, "winblend", 15)
    end
})
```

14. File System Operations

```lua
-- Using vim.fn for file operations
local exists = vim.fn.filereadable(vim.fn.expand("~/.config/nvim/init.lua"))
local is_dir = vim.fn.isdirectory("/path/to/dir")

-- Path manipulation
local full_path = vim.fn.fnamemodify("file.lua", ":p")  -- Full path
local filename = vim.fn.fnamemodify("path/to/file.lua", ":t")  -- Just filename
local dir = vim.fn.fnamemodify("path/to/file.lua", ":h")  -- Directory

-- Execute shell commands
local result = vim.fn.system("ls -la")
print(result)
```

Key Points to Remember:

1. **vim.api.nvim_*** - Low-level API functions
2. **vim.opt/vim.o** - Option management

3. **vim.keymap.set** - Modern keymap API
4. **vim.diagnostic** - LSP diagnostics
5. **vim.lsp** - LSP client functionality
6. **vim.treesitter** - Treesitter integration
7. **vim.fn** - VimL functions accessible from Lua
8. **vim.cmd** - Execute Vim commands
9. **vim.notify** - Notification system

These examples provide a solid foundation for Neovim plugin development and configuration!

Creating a Neovim Plugin with C Backend

I'll explain how to create a Neovim plugin that uses a C program as the backend, covering the architecture, communication methods, and implementation details.

Architecture Overview

```
Neovim ←→ Communication Layer ←→ C Backend Program
  ↑
Lua Plugin (Frontend)
```

Communication Methods

1. External Process (Most Common)

```lua
-- Lua plugin code
local M = {}

function M.c_backend_command(args)
    -- Spawn external C program and capture output
    local Job = require('plenary.job')

    local result = Job:new({
        command = '/path/to/your/c_program',
        args = {args},
        cwd = vim.fn.getcwd()
    }):sync()

    -- Process the result
    local output = table.concat(result, '\n')
    return output
end

-- Example usage
function M.process_file()
    local current_file = vim.fn.expand('%:p')
    local result = M.c_backend_command(current_file)
    print(result)
end
```

```
return M
```

2. Named Pipes (FIFOs)

```c
// c_backend.c - C program
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main() {
    const char* pipe_name = "/tmp/neovim_backend_pipe";

    // Create named pipe
    mkfifo(pipe_name, 0666);

    int fd = open(pipe_name, O_RDONLY);
    char buffer[1024];

    while (1) {
        int bytes = read(fd, buffer, sizeof(buffer));
        if (bytes > 0) {
            buffer[bytes] = '\0';

            // Process the command
            printf("Processing: %s\n", buffer);

            // Send response back through another pipe
            int response_fd = open("/tmp/neovim_response_pipe", O_WRONLY);
            if (response_fd != -1) {
                const char* response = "Processed successfully";
                write(response_fd, response, strlen(response));
                close(response_fd);
            }
        }
    }

    close(fd);
    return 0;
}
```

```lua
-- Lua plugin for named pipes
local M = {}

function M.send_to_c_backend(message)
    -- Write to named pipe
    local pipe = io.open("/tmp/neovim_backend_pipe", "w")
    if pipe then
        pipe:write(message .. "\n")
        pipe:close()

        -- Read response
        local response_pipe = io.open("/tmp/neovim_response_pipe", "r")
        if response_pipe then
            local response = response_pipe:read("*a")
            response_pipe:close()
            return response
        end
    end
    return nil
end

return M
```

3. Socket Communication (TCP/Unix Domain)

```c
// c_backend_server.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>

int main() {
    int server_fd, new_socket;
    struct sockaddr_un address;
    int opt = 1;
    int addrlen = sizeof(address);

    // Create socket
    server_fd = socket(AF_UNIX, SOCK_STREAM, 0);

    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "/tmp/neovim_backend_socket");

    // Remove existing socket file
    unlink("/tmp/neovim_backend_socket");

    // Bind socket
    bind(server_fd, (struct sockaddr *)&address, sizeof(address));

    // Listen for connections
    listen(server_fd, 3);

    printf("C Backend Server listening on socket\n");

    while (1) {
        new_socket = accept(server_fd, (struct sockaddr *)&address,
                           (socklen_t*)&addrlen);

        char buffer[1024] = {0};
        int valread = read(new_socket, buffer, 1024);

        // Process the command
        printf("Received: %s\n", buffer);

        // Send response
        send(new_socket, "Processed", 9, 0);
        close(new_socket);
    }

    return 0;
}
```

```
-- Lua plugin using TCP socket
local socket = require("socket")

local M = {}

function M.send_command_to_c_backend(command)
    local sock = socket.tcp()
    sock:connect("localhost", 8080)  -- or use Unix domain socket
    sock:send(command .. "\n")

    local response = sock:receive("*l")
    sock:close()

    return response
end

return M
```

Complete Plugin Example

C Backend Program

```
// text_analyzer.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    int lines;
    int words;
    int chars;
    int code_lines;
} analysis_result_t;
```

```
analysis_result_t analyze_text(const char* filename) {
    FILE* file = fopen(filename, "r");
    if (!file) {
        analysis_result_t result = {0, 0, 0, 0};
        return result;
    }

    analysis_result_t result = {0, 0, 0, 0};
    char line[1024];
    int in_multiline_comment = 0;

    while (fgets(line, sizeof(line), file)) {
        result.chars += strlen(line);
        result.lines++;

        // Count words (simple space-based)
        char* token = strtok(line, " \t\n");
        while (token) {
            result.words++;
            token = strtok(NULL, " \t\n");
        }

        // Check if it's a code line (not empty, not comment)
        char* trimmed = line;
        while (*trimmed == ' ' || *trimmed == '\t') trimmed++;
        if (strlen(trimmed) > 0 && strncmp(trimmed, "//", 2) != 0 &&
            strncmp(trimmed, "/*", 2) != 0) {
            result.code_lines++;
        }
    }

    fclose(file);
    return result;
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    analysis_result_t result = analyze_text(argv[1]);

    // Output in JSON format for easy parsing
    printf("{\n");
    printf("  \"lines\": %d,\n", result.lines);
    printf("  \"words\": %d,\n", result.words);
    printf("  \"chars\": %d,\n", result.chars);
    printf("  \"code_lines\": %d\n", result.code_lines);
    printf("}\n");
    return 0;
}
```

Lua Plugin Frontend

```lua
-- plugin/text_analyzer.lua
local M = {}

-- Compile the C program (run once during setup)
function M.compile_backend()
    local compile_cmd = "gcc -o ~/.local/share/nvim/text_analyzer text_analyzer.c"
    os.execute(compile_cmd)
end

-- Execute C backend and parse JSON response
local function execute_c_analyzer(filepath)
    local handle = io.popen(" ~/.local/share/nvim/text_analyzer " .. filepath)
    local result = handle:read("*a")
    handle:close()

    -- Simple JSON parsing (you might want to use a proper JSON library)
    local parsed = {}
    for line in result:gmatch("([^\n]+)") do
        if line:match('"lines"') then
            parsed.lines = tonumber(line:match(': (%d+)'))
        elseif line:match('"words"') then
            parsed.words = tonumber(line:match(': (%d+)'))
        elseif line:match('"chars"') then
            parsed.chars = tonumber(line:match(': (%d+)'))
        elseif line:match('"code_lines"') then
            parsed.code_lines = tonumber(line:match(': (%d+)'))
        end
    end

    return parsed
end

-- Main analysis function
function M.analyze_current_file()
    local filepath = vim.fn.expand('%:p')
    if filepath == '' then
        vim.notify("No file open", vim.log.levels.WARN)
        return
    end

    local analysis = execute_c_analyzer(filepath)

    if analysis.lines then
        local info = string.format(
            "File Analysis:\n" ..
            "Lines: %d\n" ..
            "Words: %d\n" ..
            "Characters: %d\n" ..
            "Code Lines: %d",
            analysis.lines,
            analysis.words,
            analysis.chars,
            analysis.code_lines
        )

        print(info)

        -- Optional: show in floating window
        local buf = vim.api.nvim_create_buf(false, true)
        vim.api.nvim_buf_set_lines(buf, 0, -1, false, vim.split(info, '\n'))

        local win = vim.api.nvim_open_win(buf, true, {
            relative = 'editor',
            width = 40,
            height = 8,
            row = 2,
            col = 2,
            style = 'minimal',
            border = 'rounded'
        })
    else
        vim.notify("Analysis failed", vim.log.levels.ERROR)
    end
end

-- Setup keymap
vim.keymap.set('n', '<leader>ta', M.analyze_current_file, {
    desc = "Analyze current file with C backend"
})

return M
```

Plugin Installation Script

```
-- plugin/setup.lua
local M = {}

function M.setup()
    -- Check if C program exists, compile if not
    local c_program_path = vim.fn.stdpath('data') .. '/text_analyzer'

    local file = io.open(c_program_path, 'r')
    if not file then
        -- Compile the C program
        local c_source = [[
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>

typedef struct {
    int lines;
    int words;
    int chars;
    int code_lines;
} analysis_result_t;

analysis_result_t analyze_text(const char* filename) {
    FILE* file = fopen(filename, "r");
    if (!file) {
        analysis_result_t result = {0, 0, 0, 0};
        return result;
    }

    analysis_result_t result = {0, 0, 0, 0};
    char line[1024];

    while (fgets(line, sizeof(line), file)) {
        result.chars += strlen(line);
        result.lines++;

        char* token = strtok(line, " \t\n");
        while (token) {
            result.words++;
            token = strtok(NULL, " \t\n");
        }

        char* trimmed = line;
        while (*trimmed == ' ' || *trimmed == '\t') trimmed++;
        if (strlen(trimmed) > 0) {
            result.code_lines++;
        }
    }

    fclose(file);
    return result;
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    analysis_result_t result = analyze_text(argv[1]);

    printf("{\n");
    printf("  \"lines\": %d,\n", result.lines);
    printf("  \"words\": %d,\n", result.words);
    printf("  \"chars\": %d,\n", result.chars);
    printf("  \"code_lines\": %d\n", result.code_lines);
    printf("}\n");

    return 0;
}
        ]]

        -- Write C source to temporary file and compile
        local temp_c_file = vim.fn.tempname() .. '.c'
        local f = io.open(temp_c_file, 'w')
        f:write(c_source)
        f:close()

        local compile_cmd = 'gcc -o "' .. c_program_path .. '" "' .. temp_c_file .. '"'
        local success = os.execute(compile_cmd)

        -- Clean up
        os.remove(temp_c_file)

        if success ~= 0 then
            vim.notify("Failed to compile C backend", vim.log.levels.ERROR)
            return
        end
    end
end

return M
```

Build and Distribution

Makefile for C Backend

```
# Makefile
CC = gcc
CFLAGS = -Wall -O2
TARGET = text_analyzer
SRCS = text_analyzer.c

all: $(TARGET)

$(TARGET): $(SRCS)
    $(CC) $(CFLAGS) -o $@ $<

clean:
    rm -f $(TARGET)

install: $(TARGET)
    cp $(TARGET) ~/.local/share/nvim/
```

```
.PHONY: all clean install
```

Advantages of C Backend

1. **Performance**: C programs are faster for heavy computations
2. **Existing Libraries**: Can leverage existing C libraries
3. **Memory Efficiency**: Better memory management for large data
4. **System Access**: Direct access to system resources

Considerations

1. **Compilation**: Users need a C compiler
2. **Platform Compatibility**: Different binaries for different OS/arch
3. **Distribution**: Need to handle cross-platform compilation
4. **Debugging**: More complex debugging process
5. **Security**: External process execution requires careful input validation

This architecture allows you to leverage C's performance while maintaining Lua's flexibility for Neovim integration!
